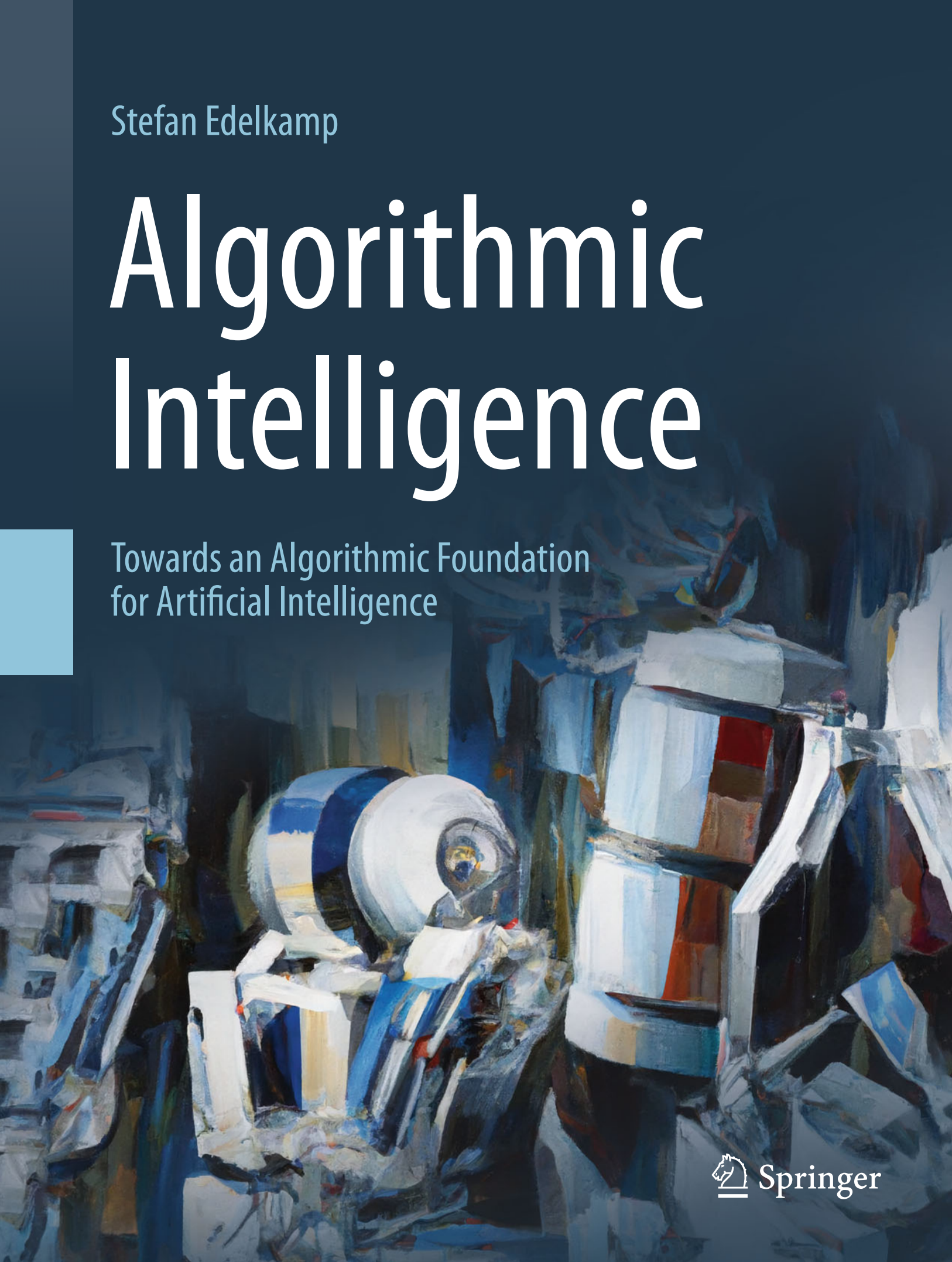
Stefan Edelkamp

Algorithmic Intelligence

Towards an Algorithmic Foundation for Artificial Intelligence



Springer

Algorithmic Intelligence

Stefan Edelkamp

Algorithmic Intelligence

Towards an Algorithmic Foundation for Artificial Intelligence

Springer

Stefan Edelkamp
AI Center, Faculty of Electrical Engineering
Czech Technical University in Prague
Prague, Czech Republic

ISBN 978-3-319-65595-6 ISBN 978-3-319-65596-3 (eBook)
https://doi.org/10.1007/978-3-319-65596-3

Cover illustration: The illustration on the cover was created by the author in DALL-E by combining keywords like algorithmic intelligence, artificial intelligence, sorting, searching, task planning, deep learning, Monte-Carlo optimization, machine learning, robot navigation, game playing, heuristic search, multiagent simulation, recommender systems, 3D printing and packing oil painting.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Dedicated to my son Lovis, and all the ones who share their love with us…

Preface

This book is concerned with *Algorithmic Intelligence*. While an algorithm can hardly be judged to be intelligent on its own, the basis of what is attributed to computer intelligence is of algorithmic roots. There is little doubt that the smaller the space consumption of an algorithm and the faster the access to its data structures are, the better the inferences.

There is a rising need for software systems capable of taking action in situations involving sensor inputs, state variables, situation assessments and environmental conditions. What we are interested in *Algorithmic Intelligence* is a holistic view encompassing areas ranging from theoretical computer science to machine learning via engineered algorithmic solutions.

Algorithmic Intelligence acknowledges the fact that computer action involves a constructive process. The machine as an information constructor is assisted by humans to create machine representations of objective reality. New information is then linked to prior knowledge. An example is a game playing program that extracts knowledge from expert play. As a result, in 2016/17, many professional Go players were defeated convincingly for the first time. In these approaches, as well as in many solutions in the book, learning is combined with search, thus combining knowledge *exploitation* with *exploration*.

The focus of the book also includes tackling optimization problems where there is no learning as such, but *Algorithmic Intelligence* comes in as a way of dealing with computational hardness. Most of the tackled problems are provably complex, so that we are more or less forced to search for the *needle in the haystack*. The mission of the book, therefore, is to promote cross-fertilization among researchers working on related optimization topics from different angles.

Algorithmic Intelligence has a core that is methodical. Artificial Intelligence has grown beyond its philosophical grounds with methods that are applied in mainstream products. With *Algorithmic Intelligence* we give this trend a name. As we avoid the term *Generalized Intelligence* with computers that generate human-like thinking, the term *Algorithmic Intelligence* relates closer to the second meaning of intelligence, as in *Business Intelligence*.

Algorithmic Intelligence stresses that modern computer designs are essential, starting from fast and atomic processor instructions up to parallelism on multiple cores. With the changes in technology, refined algorithmic designs are required. The dependence on technology advances becomes evident when looking at the very same program that has to be attributed to be more intelligent when run on a faster machine. The recent success in machine learning would not have been possible without the progress in hardware. Besides new algorithm designs like stochastic gradient descent, many- and multi-core machines helped otherwise outdated machine learning algorithms to scale, offering a fast decrease in error rates.

Algorithmic Intelligence is a pragmatic term as we agree that there is a wide divergence between the nature of machines and humans as currently understood and as revealed by conceptual analysis. Misconceptions about computer potential and misrepresentation of computer power emerge from excessive anthropomorphisation of machines. A computer program comes in a different format to what is currently understood by *human intelli-*

gence. The point is that its implemented algorithms, simple or not, can reach a high level of generalization and performance to be applicable in different contexts.

Intelligent computer programs affect everyday activities, be it gestures, spoken language inputs, or location-based services for improved navigation. We are confronted with machine recommendations in online marketplaces, and our Internet activities are correlated to map our habits. Hence, *Algorithmic Intelligence* enters our lives at different ends, e.g., in the form of phone or tablet app(lication)s, synergetic power control, or improved safety in computer networks.

The advances are practical and directly address an increased shareholder value for those that use the technology. The list of companies that have understood *Algorithmic Intelligence* to be of crucial importance to their long-term success is continuously rising.

Algorithmic Intelligence, therefore, serves as a collective term for the range of algorithmic methods that have been identified as key revenue drivers in companies (including Alphabet, Meta, Amazon, Netflix, Microsoft, UPS, and Walmart). The name opposes the term *Artificial Intelligence*, which carries the meaning that there is *intelligence*, but nothing is *real*. With *Algorithmic Intelligence* we stress the focus on the impact that well-founded algorithms already have for visible success in practice.

These pragmatics are reflected in the quote of Thomas Watson Jr.: *Our machines should be nothing more than tools for extending the powers of the human beings who use them*. The success of *Question Answering* demonstrated in playing *Jeopardy* has been ported to different fields such as cancer detection. In its *ensemble of experts* the system looks at associations among various inputs and calculates the probability that one inference method provides a better answer to a question than another and presents the top one to the user.

While *Artificial Intelligence* serves as an interface to other disciplines including Psychology and Philosophy, *Algorithmic Intelligence* aims at an interface to *Algorithm Engineering*. Suggested improvements are often provably correct, efficient, and practical, whereas Artificial Intelligence is sometimes said to cover a selection of heuristics in the form of implemented thumb rules without a theoretically sound background. With this book, we show that this does not have to be the case.

Artificial and *Computational Intelligence* are not well suited for characterizing the kind of sustainable research we are looking at in *Algorithmic Intelligence*. While the former dates back to a Dartmouth Seminar, which aimed at giving computer intelligence a name and predicted it would surpass human intelligence in the *foreseeable future*, the latter has roots in *Evolutionary Computing* and *Fuzzy Control*. Thus, the characterization of the first is seemingly too wide and the second seemingly too narrow. Researchers had to admit that computer programs have their computational requirements that have to be met, and that the encircled topics are too few to cover the recent successes.

The 21st century is the era of software that is intended to show intelligent behavior in the real world. With this book we invite all students, theoreticians and engineers interested in algorithmics to join the team. We need you to help tackle the intrinsic hardness of the problems that arise in pushing decision-making forward. Not the largest program will dictate the next level of performance, but the fastest; taking into considerations the way computers are built (including their large but finite resources). When confronted with harder problem instances, we require advanced data structures and inference methods.

This book has been written to serve as a manifestation for a steady trend, addressing current work that fits the definition we have in mind. Based on the strong interest in working solutions and application domains it is equally interesting for computer scientists in research and industry. While not aimed at being a textbook in the first place, we are certain that essentials (especially the ones with real code) are useful for teaching.

For people less interested on creating a separated new field, the subtitle *Towards an Algorithmic Foundation for Artificial Intelligence* extends the short title. It offers an alternative focus for the book with a comprehensible description of its contents. After all, the book covers many classical AI aspects like problem solving, knowledge representation and reasoning, dealing with uncertainty, machine learning, vision, and sensing and acting. It includes topics like combinatorial search, task planning, probabilistic and ontological reasoning, multiagent systems, game playing, and constraint satisfaction, which are all established research areas on their own.

In terms of machine learning, the book improves (deep) neural nets, support vector machines, nearest neighbor search, collaborative filtering, and bandit-based search algorithms. With conditional random field and tolerant pattern matching it extends reasoning with ontological background knowledge. From the algorithmic community it borrows recent hashing and priority queue data structures, as well as decision diagrams.

The book clearly does not cover everything that one may envision. A solid algorithmic foundation in AI should include further programming pearls like the bucket elimination algorithm for Bayesian network inference; inverse kinematics for robot trajectory execution; refined SMT/QBF solving for hybrid system and verification, the simplex algorithm for linear programs in constraint optimization, Lagrangian optimization for resource-bounded scheduling; negotiation and combinatorial auctions for trading; finite state machine inference for system synthesis, counterfactual regret minimization in incomplete information games, etc. This, however, would have resulted in a much larger book.

With the growing importance of the interaction with modern technology, the book emphasizes the engineering aspect. The implementation depth in the form of real and pseudo-code gradually fades out from start to the end of this book, mainly because the more realistic the application area is, the longer the code.

The topics in the book were selected in a way that the overlap to my other textbook *Heuristic Search – Theory and Applications* is kept small.

The content of this book would not exist without the work of my colleagues including Bernhard Berger, Carsten Elfers, Max Gath, Christoph Greulich, Rainer Gößl, Otthein Herzog, Shahid Jabbar, Peter Kissmann, Hartmut Messerschmidt, Damian Sulewski, Karsten Sohr, Martin Stommel, Thomas Wagner, Michael Beetz, Sabine Kuske, and David Zastrau. I am greateful to the contributions of my international collaborators Álvaro Torralba, Vidal Alcázar, Martin Dietzfelbinger, Dragan Bosnacki, Anton Wijs, Amr Elmasry, Daniele Magazzeni, Andrew and Amanda Coles, Santiago Franco, Moisés Martínez, Ionuţ Moraru, Tristan Cazenave, Samual Bounan, Lukás Chrpa, Martin Pilát, Jakub Gemrot, Andrii Nyporko, Pavel Rytír, Rostislav Horcík, Leah Chrestien, Tomás Pevný, Antonín Komenda, Jyrki Katajainen, Sebastian Wild, Stefan Schrödl, Armin Weiß, Erion Plaku, and a large number of undergraduate students. Last but not least, I want to thank my industrial collaborators Andree Lüdke, Andreas Wulfes, Ashraf Abdo, Björn Schwarze, Hendrik Rothe, Luisa Strelow, Lara Luhrmann, Salome Gindre, Tino Wahler, and Vanessa Just.

Freiburg, Dortmund, Bremen, Darmstadt, Koblenz, Paris, London, Prague *Stefan Edelkamp*

Towards a Characterization

Algorithmic Intelligence refers to the range of algorithmic methods that have been identified as key revenue drivers in companies. We are not aiming to build conscious machines; instead, we are taking advanced algorithmic techniques typically developed within *Artificial Intelligence*, and we are showing how these can be applied to decision-making problems beyond the field itself.

History and Working Groups

The term *Algorithmic Intelligence* was around before at ITU Copenhagen, Denmark by Dan Witzner Hansen, Rune Møller Jensen, Rasmus Pagh, and Georgios Yannakakis as a means of joining forces across research groups on Data Mining, Data Acquisition, Machine Learning, Computational Intelligence, Optimization, and Decision Support in order to achieve an international leading position within focus areas. In particular the aim was to achieve international visibility that can attract the best researchers in the field; develop and organize teaching programs in the area; establish a cross-group community for researchers and PhD students; establish external services for industry and institutions, facilitating knowledge transfer between fundamental insights on *Algorithmic Intelligence* and problems of high interest to society.

Similar to the famous Darthmouth Conference by John McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon, Christianizing the term *Artificial Intelligence* in 1959, the first tracable name birth-giving event was the first international workshop on *Algorithmic Intelligence* that was held on October 4th, 2011 in Berlin initiated by Stefan Edelkamp, Carsten Elfers, Rune Jensen, Hartmut Messerschmidt, and Rasmus Pagh. It kicked off with an invited talk on *Semantic Full-Text Search* by Hannah Bast. The set of topics including *A Question Answer System for Math Word Problems*, *Motion Gesture: False Positive Prediction and Prevention*, *Bitvector Full Sampling Structural Game Tree Search for Playing Cribbage*, *Signature-Inspired Security Incident and Event Management with Machine Learning Capabilities*, *New Exploration Methods in Reinforcement Learning*, *Parallel Monte-Carlo Tree Search with Pruning*, and *Weak-Heap Priority Queues in Theory and Praxis* gave us a first glimpse on what aspects *Algorithmic Intelligence* addresses.

The Internet article *Beyond AI hype: AI once stood for algorithmic intelligence* by Michael Baxter (2019) starts with

> AI this, AI that, ayeeee, I am not sure the term is always used correctly. You could say there is too much AI hype, you might as well say the tide always comes in. It is not a contentious thing to say, it just is. Maybe it would help if we re-defined it. Instead of saying AI means artificial intelligence, maybe we should return to an earlier definition, one untainted by Hollywood. Let's call it algorithmic intelligence, instead.

The term Algorithmic Intelligence has also been picked up by an interdisciplinary research center of the *Carl Zeiss Stiftung* together with Johannes Gutenberg-Universität Mainz. The so-called *Emergent Algorithmic Intelligence Center* states as objectives

> [...] the structural foundations of algorithmic intelligence are to be understood in greater depth, and thus the limits and possibilities of known machine learning methods are to be better grasped.

Problems with the Term "Artificial Intelligence"

The term *Algorithmic Intelligence* contrasts with the term *Artificial Intelligence* in two ways: Firstly, while *Artificial Intelligence* carries some suggestion that there is intelligence, but nothing real, *Algorithmic Intelligence* is focused on methods that solve a given problem. Secondly, if a method exists to solve a complex looking problem or give a good approximation of the solution, then this method belongs to *Algorithmic Intelligence*, even if it is too simple to be called *Artificial Intelligence*. While Russell and Norvig's interpretation of *Artificial Intelligence* stresses the metaphor of agents acting rationally, *Algorithmic Intelligence* prefers computers to improve decision-making capabilities of humans.

Despite breakthrough results of computers entering human life, either as robots collaboratively acting in the real-world, or recommender systems for organizing everyday activity, we avoid dwelling on the notion of *strong AI*, with computers that are able to think and capable of acting rationally in a way that humans do. We are satisfied with a much weaker and less ambitious definition of intelligence. This helps to bridge gaps to people who do not trust the information processing in a computer to be an act of human-adequate intelligence. John Kelly states in his book *Artificial Intelligence – A Modern Myth*:

> The nature of human intelligence is elusive and does not admit of a tractable formalism or effective characterization. [...] Computers are misleadingly characterized as symbol processors. In themselves they are merely repositories and manipulators of uninterpreted physical shapes, merely formal functional media. Computer systems, whether traditional hardware or software [...], are best understood as texts. Computers have no ability to act in any deep sense of the word. They do not do anything. The dependence of computers on human articulation points to fundamental limits to their potential, because of the fact that human experience, even language-mediated experience, is ultimately rooted in ineffability. Unfortunately, the deployment of explicit language conditions [...] encloses what can be thought. Misconceptions about computer potential and misrepresentation of computer power emerge from excessive anthropomorphisation of machines; the lure of an existential fallacy of the fallacy of misplaced concreteness; the list fallacy, i.e., the reliance on a mere list to provide a necessary unity; excessive respect for the power of science; belief in the ad rem adequacy of rationality; and belief in the adequacy of language.

Despite these insights, there is no doubt that there is a rising need for and rising availability of software systems capable of taking action in every-day activity, preferably operating in real-time with algorithms that adapt, generalize, assist and improve over time.

Problems with the Term "Computational Intelligence"

According to a collaborative research center with the same name, the term *Computational Intelligence* is characterized as follows.

> The field of *Computational Intelligence* (CI) covers all sorts of techniques for subsymbolic (numerical) knowledge processing, such as the well-known Fuzzy Logic (FL), Neural Networks (NN), and Evolutionary Algorithms (EA) as well as other approaches with lesser dissemination. Although CI techniques have been widely used in the last decade. The scientific goals [...] were the investigation and improvement of the foundations, applications, as well as combinations of CI methods.

Subsequently, Wikipedia provides the following definition of the term *Computational Intelligence*:

> Systems which can be seen as a part of Evolutionary Computation; *Computational Intelligence* is a set of Nature-inspired computational methodologies and approaches to address complex problems of the real world applications to which traditional (first principles, probabilistic, black-box, etc.) methodologies and approaches are ineffective or infeasible. It primarily

> includes Fuzzy logic systems, Neural Networks, and Evolutionary Computation. In addition, it also embraces techniques that stem from the above three or gravitate around one or more of them, such as Swarm Intelligence and Artificial Immune Systems which can be seen as a part of Evolutionary Computation; Dempster-Shafer Theory, Chaos Theory and Multi-valued Logic which can be seen as off-springs of Fuzzy Logic Systems, etc.

In Engelbrecht's book on *Computational Intelligence* the table of contents aligns with this restricted scope: Introduction (I), Artificial Neural Networks (II), Evolutionary Computation (III), Computational Swarm Intelligence (IV), Artificial Immune Systems (V), Fuzzy Systems (VI). The IEEE *Computational Intelligence* Society and its featured conferences (like the IEEE World Congress on *Computational Intelligence*) are also focused mainly on Fuzzy and Neural Network Computations. Other publication options like the *Journal for Computational Intelligence* use the term as a substitute for *Artificial Intelligence*.

> This leading international journal promotes and stimulates research in the field of *Artificial Intelligence* (AI). Covering a wide range of issues — from the tools and languages of AI to its philosophical implications — *Computational Intelligence* provides a vigorous forum for the publication of both experimental and theoretical research, as well as surveys and impact studies. The journal is designed to meet the needs of a wide range of AI workers in academic and industrial research.

Craenen and Eiben wrote in *Computational Intelligence; Encyclopedia of Life Support Sciences* the following glossary entry: *Computational Intelligence (CI): Subject of this introduction and a name for the combined fields of Neural Computing, Evolutionary Computation and Fuzzy Computation. Recently, the number of fields have been expanded to include DNA Computing and Quantum Computing.*

In Schwefel, Wegener and Weinert's book *Advances in Computational Intelligence* no less than 30 chapters are devoted to results achieved in *Computational Intelligence* between 1997 and 2003. The book provides a coverage of the core issues addressed in the field, especially in fuzzy logic and control as well as for evolutionary optimization algorithms including genetic programming.

In *Computational Intelligence Methods and Techniques* by Rutkowski, Springer, 2008, topics include approximate and fuzzy sets, basic structures and methods of neural networks learning, grouping of data methods, Bayesian methods, evolutionary algorithms and decision tree algorithms.

In contrast to *Computational Intelligence*, with *Algorithmic Intelligence* we stress the impact that well-founded AI algorithms have on the success in practice, while referring to a broad spectrum of efficient methods rather than centering around neural networks and fuzzy logic aspects.

Algorithmic Intelligence: Towards a Trade-off

Algorithmic Intelligence includes aspects like deep learning, game playing with incomplete information, randomized exploration with Monte-Carlo search, accelerated processing of big data, as well as advances in modern action and motion planning. It uses algorithm engineering as the technique to exploit the computational power of contemporary computer systems. Most importantly, it addresses applications areas like logistics, software model checking, and computational biology.

The term *Algorithmic Intelligence* contributes to the fact that Artificial Intelligence technologies, being a frontier discipline of Computer Science, have gradually permeated our living environments. Intelligent products have become an ubiquitous mainstay, while the goals of Algorithmic Intelligent continuously move on to meet new, exciting challenges, with a focus on technologies for a smarter world. We refer to the success stories of companies like *Google/Alphabet* or *Microsoft* with their refined Internet search, *Facebook/Meta* or *Amazon* with their collaborative filtering mechanisms, *DeepMind* with learning for the classification and prediction of pictures, language and games, *Netflix* with its recommender system for its online streaming platform, *Walmart* with its bag inspection data mining tools.

The range of applications is much larger, and through the unavoidable presence of smart mobile devices directly affects our everyday life. The focus on improvements to algorithms and data structures has left the early boots of what is currently understood under the term of *Computational Intelligence*.

With *Monte-Carlo search*, an interesting randomized search paradigm is entering the arena. Following the success story in Interactive and Combinatorial Games, it is predicted to show further advances in a widespread set of relevant industrial applications. We will see that this randomized optimization method is indeed promising and, in some cases, can produce state-of-the-art solutions to challenging problems.

Organization of the Book

This book (see following overview table) divides into four parts and several individual chapters that all provide an answer on how to combine learning with search and how to trade exploitation with exploration.

The range of different application areas gives an overview on the progress that has been made and the impact that is already there or is expected. The chapters address research ranging from Solving Puzzles, Action Planning, Game Playing, Machine Learning, via Computer Vision, Computational Biology, In- and Outdoor Logistics, Product Configuration, Urban Mobility, Video Editing, Container Packing and 3D Printing, Network Security, to Software Verification. In all these chapters the main solution method is algorithmic in nature, often combining randomized exploration with machine learning, sometimes adding background knowledge and resorting to event handling. Computer technology is addressed in looking at options for parallelization and concise memory maintenance, such as the use of disks and graphics processing unit.

Basics

At the end of the day algorithms have to be implemented in some programming language. Chapter 1 is a hands-on introduction to programming for solving combinatorial problems. The programming primer puts emphasis on recursive solutions. It is of didactic and practical value, providing insightful solutions to well-known problems. As a first set of non-trivial algorithmic designs, we look at shortest paths search in weighted state space graphs. Refined data structures for this task are the basis for solving many problems in *Algorithmic Intelligence*. In Chapter 2 priority queues are studied and cross-compared in path-finding grid benchmarks (either randomly sampled or taken from commercial games). Another building block for *Algorithmic Intelligence* are advanced sorting algorithms, introduced in Chapter 3. Reducing the number of the processor's branch misprediction will turn out to be a key concept to improve Quicksort. Hybrids of Quicksort and Mergesort further reduce the number of element comparisons, resulting in constant-factor optimal sequential sorting and in closing the gap to the lower bound. Motivated by the success in *Go*, Chapter 4 studies learning on how to play games with neural nets. Simple problems like *TicTacToe* illustrate the setup and the learning of value networks, while for advanced problems like the *SameGame* we address the learning of policy networks. Different multi-layer and convolutional network structures are compared. The results suggest that the deep ones are not always superior. Chapter 5 complements the work on learning by introducing *Monte-Carlo search*, an exploration option invoking a *myriad* of random rollouts. As one option, *Nested Rollout Policy Adaptation* compactly stores information about good successors in a vector that is updated in the course of executing the algorithm. The vector, in turn, influences the behavior of the rollout. Advanced implementation options are discussed.

Big Data

Large graphs are met in different contexts, such as social network analyses. Chapter 6 looks at traditional graph problems like finding cliques, colorings, independent sets, vertex covers, and hitting sets. Without implementation burden and domain-dependent knowledge, Monte-Carlo search yields good solutions. Chapter 7 analyzes multimedia in the form of audio-video files for improving quality in the digitalization process. Different existing tools that predict time-stamped events of artifacts within streams are correlated, increasing the prediction rate and reducing human operator cost. The general task is learning event time series for anomaly detection with fingerprints. Chapter 8 looks at events in computer networks, proposing a SIEM system with tolerant pattern matching. One question in this form of security monitoring is to find intruders in networks based on probabilistic inference with background knowledge. As an effective representative for this form of security event processing, we introduce an algorithmic scheme to combine *Conditional Random Fields* with *Ontologies*. Chapter 9 addresses machine learning for big data analysis in general and improved computer vision. It introduces a fast support vector machine that is based on the binarization of the input to overcome the *curse of dimensionality*. As a surplus the chapter explains fractal nearest neighbor search, an approximate but much faster classification algorithm of bitmap data based on Hilbert curves. Chapter 10 shows how GPS-recording devices collaboratively generate a map of the surrounding infrastructure. The incremental map generation process merges incoming traces with the set of existing ones in order to improve its own accuracy. Several filters eventually help the approach to process gigabytes of GPS data.

Research Areas

Chapter 11 looks at traditional machine learning algorithms and at options for parallelizing them on the GPU. Algorithms to be ported and accelerated include *Support Vector Machines* and matrix methods for *Collaborative Filtering*. As the global optimization scheme, we chose *Stochastic Gradient Descent*. Chapter 12 considers solving puzzles using *perfect hash functions*. The algorithmic contribution is to represent every problem state as a memory address, so that only associated information has to be stored. For domain-dependent search like games with indistinguishable or distinguishable pieces the ranking and unranking functions can be computed directly, while for unknown games we require generating perfect hash functions from the set of reachable states. Parallelization options on CPUs and GPUs are discussed. Chapter 13 provides a card game playing AI, matching the performance of top-level human players. We introduce *paranoia search* to find forced wins; we provide variants for the declarer and the opponents, and an approximation to find a forced win against most worlds in the belief space. Next, we introduce the widespread use of a minigame solver in the *factors* of the game, which restricts game play to the either trump or non-trump suit cards. The ELO ranking system has proven to be a reliable method for calculating the relative skill levels of players in zero-sum games. The evaluation of player strength in card games, however, is not obvious. We introduce an ELO system to overcome existing weaknesses. Chapter 14 addresses *action planning* with the objective to optimize the sum of action cost. We present award-winning cost-optimal PDDL planner that compactly stores exploration sets of states in binary decision diagrams. We also look at *plan recognition*. Instead of inductively starting at one initial state, in *abduction* we look for the smallest plan that completes a partial initial state to one reaching the goal. Chapter 15 progresses to *general game playing*, i.e., playing programs for games with rules that are provided as input, so that the player must play a game without knowing the game in advance just by looking at the problem description. The logical *game description language* is extended from deterministic games to games of chance with partial observability, and we present a player that can deal with this form of expressiveness. Chapter 16 considers a multiagent simulation solution for in- and outdoor navigation problems. We look at an event-driven simulation system which has been designed to solve and evaluate scenarios of the logistics domain. In particular the system allows to simulate autonomous logistic processes where autonomous agents perform planning and decision processes. Chapter 17 looks at the problem of product configuration as a complex constraint satisfaction problem. The configuration space is tremendously large and besides the continued reduction of configuration choices of the

user by constraint propagation, this operation can be learnt, and we suggest likely fitting product parameters. A hybrid of case-based reasoning and rule mining leads to a compromise between a high number of correct predictions and a low number of incorrect predictions.

Application Areas

Chapter 18 looks at *adversarial planning*. Effective decision-making in adversarial environments is important for many real-world applications. The presence of an adversary often strongly influences the agent's ability to achieve its goals. Automated planning, however, often restricts to static environments, where only one agent applies its actions. While such techniques are effective, the presence of an adversary strongly influences plan quality, so that plan generation has to take into account possible actions of competing agents. In other words, the agent should know what the other agents will likely do. Chapter 19 studies the automated verification of computer programs via software *model checking*. We will show how this general state-space exploring task can be ported on the GPU to accelerate the verification process to either report a bug or the correctness with respect to a given specification. We will see that a certain combination of multi-core CPU and many-core GPU is most effective. Chapter 20 considers solving the *holy grail* in Computational Biology, the *multiple sequence alignment problem*, for which sequences of DNA (or protein) strings are to be aligned so that the similarity score is maximized. The problem relates to approximate string matching. As dynamic programming solutions suffer from limited amounts of main memory, randomized search improves already existing solutions. Chapter 21 considers a practical problem of groupage routing with time windows and premium services in a multiagent system. It also looks at packing hundreds of boxes for a disassembled object (e.g., a car) into as few containers as possible, while satisfying additional packing constraints. Once shipped, the object can be unpacked, assembled, and sold. 3D Packing with object orientation is a challenging optimization problem for which, again, randomized search and learning will be the keys towards a solution. Chapter 22 looks at packings for objects of irregular shape. The task arises in the additive manufacturing in a 3D printer, where—in order to save time—several object parts are produced in parallel. The sphere-tree representation of the objects helps to detect collisions fast and to measure the according overlap of the intersections. These methods are then included in a global optimizer based on simulated annealing. Chapter 23 tackles multi-goal motion planning like the *physical traveling salesman problem* to perform freespace navigation for a moving robot with velocity and acceleration in a bitmap environment of obstacles. We then go one step further and look at 2D and 3D mesh environments to be inspected with complex vehicle models and solved with filtering skeletons followed by a cyclic interplay between a sampling-based motion planning step in the continuous space, and by calling a TSP solver in the discrete space. Chapter 24 looks at flow production of goods. In particular, we examine a monorail infrastructure with assembly stations. To compute optimized solutions the physical setting is modeled in a cyber-physical discrete-event simulator. The simulation is then further simplified to find improved schedules for the shuttles. We adapted a model checker to drive the exploration. Finally, in Chapter 25 we look at further possible application areas that reflect some avenues for algorithmic intelligence. Last, but not least, we provide an extensive index and a list of references for further reading.

Chapter	Title	Methods (selection)	Domains (selection)	Data Structures (selection)
1	Programming Primer	Recursion, Branch-and-Bound, Backtracking	Sudoku, 15-Puzzle, Peg Solitaire, TSP, Towers-of-Hanoi, *n*-Queens, Number Partitioning, Mastermind, MisterX	Arrays, Records
2	Shortest Paths	Dijkstra's Shortest Path Algorithm, A*, Flood-Fill	Weighted Graphs, Grids	*k*-ary, Radix, Fibonacci, Pairing, Factorized Heaps, Bucket Maps
3	Sorting	Quick-, Merge-, Heapsort, MergeInsertion, Blocked-Quicksort, Strong-Heapsort, QuickMergesort	Ordered Sets, Numbers	Vectors, Fine, Weak, Strong, Lazy Strengthened Heaps
4	Deep Learning	Backpropagation, Retrograde Analysis, Stochastic Gradient Descent	One- and Two-Player Games	Arrays, Multilayered/Convolutional Policy/Value Neural Nets
5	Monte-Carlo Search	Nested Monte-Carlo Search, NRPA, UCT	SameGame, Snake-In-Box, Vehicle Routing	Arrays, Lists
6	Graph Data	SAT Solving, Monte-Carlo Search	Clique, Independent Set, Vertex Cover, Graph Coloring, Hitting Set	Adjacency Matrix
7	Multimedia Data	Feature Selection and Extraction, Random Forest	Digital Tapes	Interval Tree, Event Matrix, Fingerprint
8	Network Data	Tolerant Pattern Matching, Anomaly Detection, Symbolic Aggregation, Sensor Fusion, Improved Iterated Scaling	KDD Benchmark, Industrial Data	Conditional Random Field, Ontology
9	Image Data	Fractal Nearest Neighbor Search, Bitvector Machine	Computer Vision Bechmarks	Arrays, Van-Emde Boas PQ
10	Navigation Data	Mapping, Skeletonization via Morphologic Operations	GPS Traces	Raster Maps, Trace/Tile DB, Cache
11	Machine Learning	Support Vector Machine, Collaborative Filtering, Stochastic Gradient Descent	Various ML Benchmarks	Bitmaps, SIFT/SURF Vectors
12	Problem Solving	Perfect Hashing, Ranking, Bitvector Search Symbolic Classification	Selection & Permutation Games, Connect-Four	Array, BDDs
13	Card Game Playing	AND/OR-, Paranoia-, and Factorized Search	Skat, Games of Chance	Statistical Tables, Knowledge Vectors
14	Action Planning	Symbolic A*/Bidirectional Dijkstra/Abduction	Benchmarks IPC	Arrays of BDDs
15	General Game Playing	Retrograde Analysis, Belief Set Search	Game Description Language Games	BDDs, Information Sets
16	Multiagent Systems	Simulation, Single-Source Shortest Path Search, Dynamic Programming	Open Street Map, Public Transportation Data	Time-Expanded/-Dependent Graph
17	Recommender Systems	Apriori, FP Growth, *k*-Nearest Neighbor Search	Industrial Records	Decision Lists/Trees, FP-Tree
18	Adversarial Planning	Double-Oracle, Best-Response, Benchmarks	Nash Equilibrium Matrices	
19	Model Checking	Bloom Filtering, Rewriting in Reverse Polish Notation	Verification Bechmarks	Bitvector, Open/Closed Lists, Vectors
20	Computational Biology	Iterative-Deepening Dynamic Programming, NRPA	Sequence Alignment Benchmarks	Arrays, Lists
21	Logistics	Monte-Carlo Search, Axis-Aligned Box Intersection	Squares, Industrial Car Packings	Arrays
22	Additive Manufacturing	Medial Axis Computation, Simulated Annealing	CAD Benchmarks	Voronoi Diagram, Sphere Tree, Packing, Octree
23	Robot Motion Planning	Single-Source Shortest Paths Search, TSP Solving, Branch-and-Bound Search, Grassfire	Physical TSP Benchmarks, 2D/3D Motion Planning Problems	Grid Graphs, Radix Heaps
24	Industrial Production	Local Virtual Time / Discrete Event System Simulation, Constraint Branch-and-Bound	Monorail Manufacturing System	Event List
25	Further Applications	Temporal Pattern Mining, Rule Induction	Pedictive Maintenance, Data Imputation	Time Series, Trees, Graphs

Contents

Part I Basics 1

1 Programming Primer 3
- 1.1 Recursion . . . 4
  - 1.1.1 Divide-and-Conquer . . . 5
  - 1.1.2 Recursion on Texts . . . 5
  - 1.1.3 Factorial Numbers . . . 6
  - 1.1.4 Fibonacci Numbers . . . 6
  - 1.1.5 Ackermann Numbers . . . 7
  - 1.1.6 Ulam Numbers . . . 8
- 1.2 Calculus . . . 8
  - 1.2.1 Square Roots . . . 8
  - 1.2.2 Euclid's Algorithm . . . 9
  - 1.2.3 Pascal's Triangle . . . 9
  - 1.2.4 Prime Factorization . . . 10
  - 1.2.5 Gaussian Elimination . . . 10
  - 1.2.6 Min-Max Problem . . . 12
  - 1.2.7 Quickselect and Quicksort . . . 12
- 1.3 Backtracking . . . 13
  - 1.3.1 Post's Correspondence Problem . . . 14
  - 1.3.2 Towers-of-Hanoi . . . 15
  - 1.3.3 Mazes . . . 15
  - 1.3.4 The Queens Problem . . . 16
  - 1.3.5 Sudoku . . . 17
- 1.4 Heuristic Search . . . 18
  - 1.4.1 Number Partitioning . . . 19
  - 1.4.2 The 15-Puzzle . . . 19
  - 1.4.3 Ranking and Unranking . . . 22
  - 1.4.4 Peg Solitaire . . . 22
  - 1.4.5 Traveling Salesman Problem . . . 24
- 1.5 Randomization . . . 26
  - 1.5.1 Randomized Prime Number Tests . . . 26
  - 1.5.2 Mister X . . . 27
  - 1.5.3 Mastermind . . . 28
  - 1.5.4 Nim . . . 29
  - 1.5.5 Snake . . . 30
  - 1.5.6 PacMan . . . 31
- 1.6 Bibliographic Notes . . . 33

2 Shortest Paths 35
- 2.1 Introduction . . . 35
- 2.2 Dijkstra's Algorithm . . . 36
- 2.3 General Priority Queues . . . 37

2.3.1 *k*-ary Heaps . . . 38
2.3.2 Fibonacci Heaps . . . 38
2.3.3 Pairing Heaps . . . 38
2.4 Bucket Priority Queues . . . 39
2.4.1 Radix Heaps . . . 40
2.4.2 Bucket Maps . . . 41
2.4.3 Factorized Heaps . . . 41
2.5 Cache-Efficient Flood-Filling . . . 43
2.6 Experiments . . . 45
2.7 Summary . . . 45
2.8 Bibliographic Notes . . . 46

3 Sorting 49
3.1 Introduction . . . 49
3.2 Heapsort . . . 50
3.3 Strong Heapsort . . . 50
3.3.1 Strong Heap Construction . . . 51
3.3.2 Sorting with Strengthened Heaps . . . 54
3.4 Improving Quicksort . . . 57
3.5 Block Quicksort . . . 57
3.6 Quick Mergesort . . . 59
3.7 Summary . . . 64
3.8 Bibliographic Notes . . . 65

4 Deep Learning 67
4.1 Introduction . . . 67
4.2 Case Study: TicTacToe . . . 68
4.3 Case Study: Same Game . . . 72
4.4 Case Study: Sokoban . . . 74
4.4.1 Designing Loss Functions . . . 78
4.4.2 Definition of L* . . . 79
4.5 Greedy Best-First Search: Optimizing Rank . . . 83
4.6 Summary . . . 83
4.7 Bibliographic Notes . . . 84

5 Monte-Carlo Search 85
5.1 Introduction . . . 85
5.2 Monte-Carlo Search . . . 86
5.2.1 Upper Confidence Bounds Applied to Trees . . . 86
5.2.2 Parallel Monte-Carlo Search . . . 88
5.2.3 Nested Monte-Carlo . . . 90
5.2.4 Nested Rollout Policy Adaptation . . . 90
5.3 Beam NRPA . . . 92
5.4 Refinements . . . 93
5.5 Improving the Diversity . . . 96
5.5.1 Improving Diversity in the NRPA Driver . . . 96
5.5.2 Improving Diversity in the Policy Adaptation . . . 97
5.6 Case Study: SameGame . . . 98
5.7 Case Study: Snake-in-the-Box . . . 100
5.8 Case Study: Vehicle Routing . . . 101
5.9 Summary . . . 103
5.10 Bibliographic Notes . . . 104

Part II Big Data **105**

6 Graph Data **107**
6.1 Introduction . . . 107
6.2 Mathematical Encoding . . . 108
6.3 PDDL Encodings . . . 108
6.4 SAT Encodings . . . 109
6.5 Game Encodings . . . 110
6.6 Experiments . . . 112
6.6.1 Clique . . . 113
6.6.2 Graph Coloring . . . 114
6.6.3 Independent and Hitting Sets . . . 114
6.7 Summary . . . 115
6.8 Bibliographic Notes . . . 117

7 Multimedia Data **119**
7.1 Introduction . . . 119
7.2 Problem Formulation . . . 120
7.3 Feature Extraction . . . 121
7.4 Evaluation . . . 125
7.5 Summary . . . 126
7.6 Bibliographic Notes . . . 128

8 Network Data **129**
8.1 Introduction . . . 129
8.2 Security Information and Event Management . . . 131
8.3 Anomaly Detection . . . 132
8.4 Tolerant Pattern Matching with Background Knowledge . . . 136
8.4.1 Preliminaries . . . 136
8.4.2 Tolerant Pattern Matching . . . 136
8.4.3 Divide-and-Conquer . . . 138
8.4.4 Complexity Considerations . . . 141
8.4.5 Conditional Random Fields with Tolerant Features . . . 142
8.5 Efficiency of Incident Detection . . . 146
8.5.1 Experiments with ArcSight . . . 146
8.5.2 Simulation Experiments . . . 148
8.6 Summary . . . 150
8.7 Bibliographic Notes . . . 150

9 Image Data **153**
9.1 Introduction . . . 153
9.2 Nearest Neighbor Computation . . . 155
9.3 Support Vector Machines . . . 156
9.3.1 Bitvector Machine . . . 157
9.3.2 Case Studies . . . 158
9.4 Experiments . . . 159
9.5 Summary . . . 163
9.6 Bibliographic Notes . . . 164

10 Navigation Data **165**
10.1 Introduction . . . 165
10.2 Map Generation . . . 167
10.3 Map Administration . . . 169

10.4 Extraction of Road Surfaces . . . 170
10.5 Road Skeleton Computation . . . 172
10.5.1 Skeleton Generation . . . 172
10.5.2 Skeleton Minimization . . . 173
10.6 Graph Construction . . . 174
10.7 Integration to a Traffic Simulator . . . 176
10.7.1 Post-processing of the SUMO Street Network . . . 176
10.7.2 Traffic Routes . . . 177
10.8 Summary . . . 177
10.9 Bibliographic Notes . . . 178

Part III Research Areas **179**

11 Machine Learning **181**
11.1 Introduction . . . 181
11.2 Machine Learning . . . 182
11.3 Neural Networks . . . 183
11.4 GPGPU Essentials . . . 185
11.5 Iterative Gradient Descent and Parallelization . . . 187
11.6 Applications . . . 188
11.7 Summary . . . 191
11.8 Bibliographic Notes . . . 191

12 Problem Solving **193**
12.1 Introduction . . . 193
12.2 Perfect Hashing . . . 195
12.3 Bitvector State Space Search . . . 195
12.4 Hashing Permutation Games . . . 196
12.5 Hashing Selection Games . . . 198
12.6 Parallelization . . . 200
12.7 Experiments Explicit-State Perfect Hashing . . . 202
12.8 Binary Decision Diagrams for Strongly Solving Games . . . 208
12.9 Experiments BDD Hashing . . . 213
12.10 Bibliographic Notes . . . 213

13 Card Game Playing **217**
13.1 Introduction . . . 217
13.2 Bidding . . . 220
13.3 Open Game Play . . . 221
13.4 Skat Bot Architecture . . . 224
13.5 Knowledge-based Paranoia Search . . . 225
13.6 Factorized Search . . . 230
13.6.1 The Power of Suit Partitioning . . . 230
13.6.2 Miniglassbox Search . . . 231
13.6.3 Single-Dummy Miniglassbox Solver . . . 231
13.6.4 Double-Dummy Miniglassbox Solver . . . 232
13.7 World Search . . . 234
13.8 Open-Card Solver Voting . . . 235
13.9 Hope Cards . . . 236
13.10 Towards an ELO Rating System . . . 236
13.10.1 Scoring Systems and Hand Strength . . . 238
13.10.2 ELO System for Zero-Sum Games . . . 239

13.10.3 ELO System for Games of Chance . . . 240
13.11 Experiments . . . 242
13.12 General Imperfect Information MiniMax Search . . . 247
13.12.1 Perfect-Information Monte Carlo . . . 248
13.12.2 AlphaMu . . . 249
13.12.3 Permutation . . . 250
13.13 Summary . . . 253
13.14 Bibliographic Notes . . . 254

14 Action Planning **257**
14.1 Introduction . . . 257
14.2 Symbolic Search . . . 258
14.3 Heuristic Search Planning . . . 260
14.3.1 Pattern Database . . . 260
14.3.2 Merge-and-Shrink . . . 260
14.4 Symbolic A* Search . . . 261
14.4.1 Basic Improvements . . . 262
14.4.2 List BDDA* . . . 263
14.5 Symbolic Merge-and-Shrink . . . 263
14.5.1 ADD Complexity . . . 264
14.5.2 Limits and Possibilities . . . 264
14.6 Comparison . . . 266
14.7 Complementary . . . 267
14.7.1 Configuration Choices . . . 269
14.7.2 Results . . . 270
14.8 Symbolic Abduction . . . 271
14.8.1 Computing Valid Hypotheses . . . 272
14.8.2 Uniform-Cost Abductive Inference . . . 272
14.8.3 Cost-Optimal Abductive Inference . . . 273
14.8.4 Manual Selection Strategies . . . 274
14.8.5 Finding Critical Query Variables . . . 275
14.9 Plan Recognition Experiments . . . 275
14.10 Summary . . . 276
14.11 Bibilographic Notes . . . 278

15 General Game Playing **281**
15.1 Introduction . . . 281
15.2 General Game Playing . . . 282
15.2.1 GDL . . . 282
15.2.2 GDL-II . . . 283
15.3 Solving General Two-Player Turn-Taking Games . . . 285
15.4 Handling Incomplete Information . . . 290
15.4.1 Full Set of Belief States . . . 290
15.4.2 Subset of Belief States . . . 290
15.4.3 Choosing a Move for a Set of Belief States . . . 293
15.5 Experiments . . . 294
15.6 Summary . . . 295
15.7 Bibliographic Notes . . . 295

16 Multiagent Systems **297**
16.1 Introduction . . . 297
16.2 Multiagent-based Simulation . . . 298
16.3 Simulation Model . . . 298

16.4 Shortest Path Search . . . 301
16.5 Experimental Setup . . . 301
16.6 Current Status and Findings . . . 306
16.7 Summary . . . 307
16.8 Bibliographic Notes . . . 308

17 Recommendation and Configuration 309
17.1 Introduction . . . 309
17.2 Recommender Systems . . . 310
17.3 Binarized Clustering . . . 312
17.4 Product Configuration . . . 313
17.5 Recommendations for Product Configuration . . . 315
17.6 Algorithms . . . 315
17.6.1 Association Rule Mining . . . 315
17.6.2 Nearest Neighbor . . . 316
17.6.3 Integration . . . 318
17.7 Evaluation . . . 320
17.8 Summary . . . 320
17.9 Bibliographic Notes . . . 322

Part IV Applications 323

18 Adversarial Planning 325
18.1 Introduction . . . 325
18.2 Basics in Game Theory . . . 326
18.2.1 Nash Equilibria . . . 326
18.2.2 Double-Oracle Algorithm . . . 327
18.3 Incorporating Planning into the Double Oracle Algorithm . . . 328
18.4 Cost-Adversarial Planning Games . . . 330
18.5 Experiments . . . 332
18.5.1 Temporal Planning Domains . . . 332
18.5.2 Cost-Adversarial Planning Games . . . 333
18.6 Summary . . . 333
18.7 Bibliographic Notes . . . 334

19 Model Checking 337
19.1 Introduction . . . 337
19.2 GPU Essentials . . . 338
19.3 Probabilistic Model Checking . . . 339
19.4 GPU Breadth-First Search . . . 340
19.4.1 Preparing the Model . . . 340
19.4.2 State Exploration on the GPU . . . 343
19.5 Experiments . . . 345
19.6 Summary . . . 347
19.7 Bibliographic Notes . . . 348

20 Computational Biology 349
20.1 Introduction . . . 349
20.2 Monte-Carlo Tree Search for MSA . . . 351
20.2.1 Construction of all Alignment Columns . . . 351
20.2.2 Construction of all Alignment Gaps . . . 353
20.2.3 Construction of an Initial Alignment . . . 354

20.3 Experimental Results . . . 356
20.4 Summary . . . 358
20.5 Bibliographic Notes . . . 359

21 Logistics **361**
21.1 Introduction . . . 361
21.2 Dispatching in Groupage Traffic . . . 363
21.2.1 Constraint ATSP Solving . . . 365
21.2.2 Agent Dispatching and Simulation . . . 367
21.3 Container Packing with Nested Monte-Carlo Search . . . 368
21.3.1 Extreme Points . . . 368
21.3.2 Box Packing . . . 369
21.4 Experiments . . . 371
21.4.1 Groupage Traffic . . . 371
21.4.2 Packing . . . 373
21.5 Summary . . . 374
21.6 Bibliographic Notes . . . 375

22 Additive Manufacturing **377**
22.1 Introduction . . . 377
22.2 Sphere-Tree Construction . . . 377
22.3 Robustness Considerations . . . 380
22.4 Global Optimization . . . 381
22.5 Experimental Results . . . 383
22.6 Summary . . . 385
22.7 Bibliographic Notes . . . 386

23 Robot Motion Planning **387**
23.1 Introduction . . . 387
23.2 RRT, RRT*, and Deep RRT* . . . 388
23.3 Physical TSP . . . 390
23.4 Shortest Paths . . . 391
23.5 TSP Search . . . 393
23.5.1 Optimal TSP Solving . . . 393
23.5.2 Suboptimal TSP Solving . . . 397
23.6 Inspection Problem . . . 398
23.7 Method . . . 399
23.7.1 Generating the Inspection Points . . . 399
23.7.2 CTSP Solver . . . 401
23.7.3 Following the CTSP Tour . . . 401
23.8 Evaluation . . . 402
23.8.1 PTSP Simulation . . . 402
23.8.2 Robot Simulation . . . 403
23.8.3 2D Inspection . . . 404
23.8.4 3D Inspection . . . 405
23.9 Summary . . . 406
23.10 Bibliographic Notes . . . 408

24 Industrial Production **411**
24.1 Introduction . . . 411
24.2 Preliminaries . . . 412
24.2.1 Discrete Event Simulation . . . 412
24.2.2 Flow Manufacturing . . . 414

24.3 Case Study . . . 416
24.4 Promela Specification . . . 417
24.5 Optimized Scheduling . . . 420
24.5.1 Guarded Branching . . . 421
24.5.2 Process Synchronization . . . 421
24.6 Game Encoding . . . 423
24.7 Evaluation . . . 425
24.8 Summary . . . 426
24.9 Bibliographic Notes . . . 427

25 Further Application Areas **429**
25.1 Introduction . . . 429
25.2 Vacancy Simulation and Temporal Pattern Mining in Smart Homes . . . 432
25.3 Form Filling . . . 433
25.4 Demand and Sales Prediction for Retail Trade . . . 433
25.5 Geo-Location Tagging for Better Goods Assortment . . . 433
25.6 Automated Inventory List Creation with Vision and Speech . . . 433
25.7 Temporal Task-Motion Planning for Robots with Resources . . . 434
25.8 Optimizing Integrated Production and Route-Planning . . . 434
25.9 Safe Travel Recommendation with Movement Data . . . 434
25.10 Improved Static Code Analysis with Machine Learning . . . 435
25.11 Contract Prediction on Account Transaction Series . . . 435
25.12 Recommender Systems for eCommerce . . . 435
25.13 Predictive Maintenance to Produce Assemblies . . . 436
25.14 Knowledge for Retail . . . 436
25.15 Industrial Inspection . . . 436
25.16 Skill Recommendation . . . 436
25.17 Bibliographic Notes . . . 437

Index and References **439**
Index . . . 439
References . . . 449

Part I
Basics

Chapter 1
Programming Primer

This chapter is a hands-on introduction to programming for solving combinatorial AI problems. The programming primer puts emphasis on recursive solutions. It is of didactic and practical value, providing insightful solutions to priority queue well-known problems.

Object orientation helps handling larger software projects. For Niklaus Wirth it is the consequent continuation of structuring code with elements like methods and records. The success of object-oriented industrial software development is inevitable. Some software engineers propose to simply highlight verbs in a text that describes the solution of the task in natural language, and, from there, extract objects and methods. However, there are limits, as for adding two numbers it is an overkill having to say to the first one that it has to cumulate itself to the second one.

There is no doubt that object orientation should be taught to programmers, but opinions diverge on when to do this. *Object First* purposes to start from the very beginning, and to put the concept of a class in the center of teaching on how to program. In Java all source code already is encapsulated in classes, while *Object First* goes further and proposes to teach object-orientation with the very first program, before other concepts like algorithms and data structures are introduced. Decker and Hirschfeld write: *To be sure, we struggled long and hard with each of these [reasons for misunderstanding], but have since come to recognize them all to reflect one or more of the following: our lack of understanding of the paradigm, our fear of the commonly-used object-oriented programming languages (most notably C++), our procedural biases derived from years of teaching Pascal, or what we now see as a growing body of object-oriented programming that reflects a number of myths about object-oriented programming*. There are many reasons for introducing objects early, but when *Objects First* ends up in *No Algorithms*, something has gone wrong.

What programming language to use? ETH takes *C*, others use *Python*, *Lua*, or even *MatLab*. For children, *Scratch* has been proposed. According to the TIOBE Index in programming practice C is the most common choice, followed by Java, Objective C, C++ and C#. Together they take a share of more than 60% of all written programs. Imperative programming is a long-standing base with similar syntax in most of the widespread languages. As a result, we used the imperative kernel of *Java* to get started in programming intelligence, switching to C and pseudo codes later on.

We assume that first contacts with types like `char`, `int`, `long`, `float` and `double`, with loops like `while(.) {.}`, `for(.;.;.) {.}`, bzw. `do {.} while(.);`, with branches like `if (.) {.} else {.}`, bzw. `(.) ? {.} : {.}`, with arrays like `int a[..]` and with methods ending in `return` have been made.

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_1

1.1 Recursion

Abbreviations like GNU for *GNU's not UNIX*, or *mise en abyme* artifacts like the illustrations of M.C. Escher, the book *The never-ending story* by Michael Ende, the infinite reflections on parallel mirrors, the film *Matrix*, or the unpacking of Matryoshkas are well-known examples of *recursion*.

Fractals like the *Julia set* or *Koch's, Peano's* and *Hilbert's curves* are created through the iterated application of recursive equations. The *Mandelbrot Set* is defined as complex numbers c, for which the iterated evaluation of $z_{n+1} = z_n^2 - c$ with $z_0 = 0$ converges. Using the representation of numbers with real and imaginary part $(r_0, i_0) = (0,0)$ and some vector (c_r, c_i) links it to the iterated computation of $r_{n+1} = r_n^2 - i_n^2 + c_r$ and $i_{n+1} = 2 r_n i_n + c_i$. The following program is not recursive, while the definition of the set is.

Program 1.1: Program for computing the Mandelbrot set.

```
public class Mandelbrot
{
    /**
     * Computes and prints Mandelbrot fractal on screen
     *
     */
    public void fractal() {
      int k = 2, double y = -16; // x,y are coordinates
      String magic = new String(" .:-;!/>)|iIHO+");
      while (y++<15) { // r,i are updated
        String row = new String();
        for (double x=0;x<84;x++) {
          row = row + magic.charAt(k&15);
          k = 0;
          double imag = 0.0, real = 0.0, j;
          do {
            j = real*real - imag*imag -2 + x/25.0;
            imag = 2*real*imag + y/10; real = j;
          } while(j*j+imag*imag<11 && k++<111);
        }
        System.out.println(row);
      }
    }
}
```

The output is as follows.

1.1.1 Divide-and-Conquer

Instead of saving an amount of $x = 2^k$ alone, Program 1.2 applies to *divide-and-conquer* and recursively asks two friends to contribute half of the amount each.

Program 1.2: Divide-and-Conquer.

```
public class CollectMoney
{
    int total;
    /**
     * Divide and conquer method to collect money from friends
     */
    void dac(int a) { // recursive collect function
      if (a<=1) total += a; // increase total amount
      else { dac(a/2); dac(a/2); } // recursive calls
    }
    int collect(int a) { total = 0; dac(a); return total; }
}
```

1.1.2 Recursion on Texts

Recursion does not only work on numbers but also on strings. Given the input `hallo.` Program 1.3 returns `.ollah`. By a slight modification we get the *palindrome* `hallo.ollah`.

Program 1.3: Reversal of a string and building a palidrome.

```
public class Reversal
{
    /**
     * Printing an input string in reverse order
     */
    public void reverse(String s, int i) {
       if (s.charAt(i) != '.') reverse(s,i+1);
       System.out.print(s.charAt(i));
    }
    /**
     * Printing an input string as palindrome
     */
    public void palindrome(String s, int i) {
       System.out.print(s.charAt(i));
       if (s.charAt(i) != '.') palindrome(s,i+1);
       System.out.print(s.charAt(i));
    }
}
```

1.1.3 Factorial Numbers

Suited to induction, several mathematical functions have a recursive formulation. For example, the *factorial* of n is the number of permutations of n pairwise distinct objects. The definition $n! = 1 \cdot \ldots \cdot n = n \cdot (n-1)!$ with $0! = 1$ leads to Program 1.4.

Program 1.4: Computing the number of permutations.

```
public class Factorial
{
    /**
     * Computes number of permutations of n objects
     */
    public long f(long n) {
      return (n==0) ? 1 : n*f(n-1);
    }
}
```

1.1.4 Fibonacci Numbers

The *Fibonacci* numbers are defined by the recursion $f(n) = f(n-1) + f(n-2)$ with $f(0) = 1$ and $f(1) = 1$. There is a direct implementation shown as f in Program 1.5, and two faster ones: *fiter* that uses *Dynamic Programming*, and *fib* that applies *memoization*.

Program 1.5: Fibonacci's functions.

```
public class Fibonacci
{
    /**
     * Compute Fibonacci numbers in different ways
     */
    public int f(int n) {
        return (n<=1) ? 1 : f(n-1) + f(n-2);
    }
    public int [] F = new int[100];
    public int fib(int n) {
      if (n<=1) return 1;
      if (F[n]!=0) return F[n];
      F[n-1] = fib(n-1);
      return fib(n-2) + fib(n-1);
    }
    public int fiter(int n) {
      F[0] = F[1] = 1;
      for (int i = 2; i<=n; i++)
        F[i] = F[i-1] + F[i-2];
      return F[n];
    }
}
```

1.1.5 Ackermann Numbers

The Ackermann function is an even faster growing function. For $n = 0$ we have $a(n,m) = m+1$, for $m = 0$ we have $a(n,m) = a(n-1,1)$ and $a(n,m) = a(n-1,a(n,m-1))$, otherwise. The function was proposed 1926 by Wilhelm Ackermann and has shown limits to some computational models.

Program 1.6: Ackermann's function.

```
public class Ackermann
{
    /**
     * Evaluate Ackermann function
     */
    public int a(int n, int m) {
      return (n == 0) ? m+1 : (m == 0) ? a(n-1,1) : a(n-1,a(n,m-1));
    }
}
```

Similarly, the *two tower* $x_n = 2^{x_{n-1}}$ with $x_0 = 1$ is a fast-growing function.

Program 1.7: Two-tower function.

```
int to(int n, int k) { return (k == 0) ? 1 : n * to(n,k-1); }
int tower(int n) { return (n == 1) ? 2 : to(2,tower(n-1)); }
```

1.1.6 Ulam Numbers

The following function (Program 1.8) u was first described by Stanislaw Marcin Ulam: for $x = 1$ we have $u(x) = 1$, for $x > 1$ and even x we have $u(x) = u(\lfloor x/2 \rfloor)$, otherwise $u(x) = 3x+1$. The value of Ulam's function is always 1 and should not be mixed with Ulam's series. Some inputs like 113,383 may exceed the computer's integer number representation.

Program 1.8: Ulam's function.

```
public class Ulam
{
    /**
     * The Ulam sequence
     */
    public int u(int n) {
      System.out.print("u("+n+"),");
      return n == 1 ? 1 : n%2 == 0 ? u(n/2) : u(3*n+1);
    }
}
```

1.2 Calculus

1.2.1 Square Roots

Program 1.9 computes the square root of the input number following Heron's algorithm. The computation is bound to integers, leaving the fractional part out. We think of a rectangle, of which one side is the number of which we want to compute the square root and the other number is 1. The algorithm iteratively transforms the rectangle into a square of the same area. The implementation uses two integer variables `a` and `b`; if they are not next to each other, then the mean of the two is computed for `a`, while `b` is chosen to maintain the resulting area.

Program 1.9: Iterative square roots determination.

```
public class SquareRoot
{
    public int method(int i)
    {
        int a = i, b = 1;
        while (a - b > 1) { a = (a + b) / 2, b = i / a; }
        return (a + b) / 2;
    }
```

1.2.2 Euclid's Algorithm

The recursive computation of the *greatest common divisor* (GCD) is as follows. If $a \bmod b = 0$, then $gcd(a,b) = b$, otherwise $gcd(a,b) = gcd(b, a \bmod b)$. The parameters are selected so that the second number is always smaller than the first one. *Euclid's algorithm* is efficient, as in every second step the input number is at least halved. The *least common multiplier* (LCM) of a and b is ab divided by the GCD. Based on this observation, Program 1.10 computes the reduced sum of two fractions.

Program 1.10: Computation with reduced fractions.

```
public class Fractional
{
    /**
     * Euclid's algorithm to compute greatest common divisor of a and b
     */
    public int gcd(int a, int b) { return a == 0 ? b : gcd(b%a,a); }
    /*
     * Compute least common multiplier of a and b
     */
    public int lcm(int a, int b) { return a*b / gcd(a,b); }
    public void test(int a, int b, int c, int d) {
      int k = lcm(b,d), s = a*k/b + c*k/d;
      System.out.println("sum = ("+(s/gcd(s,k))+"/"+(k/gcd(s,k))+")");
    }
}
```

1.2.3 Pascal's Triangle

The Binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is used e.g. in $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$. We have

$$c(n,k) = \binom{n}{k} = \frac{n!}{(n-k)! \cdot k!} = \frac{n(n-1)!}{k(n-k)!(k-1)!} = \frac{n}{k} c(n-1,k-1)$$

with $c(n,0) = 1$. Program 1.11 outputs Pascal's triangle: $\binom{n}{k}$ for $n \leq m$ and all $k = 0, \ldots, n$.

Program 1.11: Pascal's triangle.

```
public class Binomial
{
    /**
     * Computes Binomial numbers n choose k
     */
     private int choose(int n, int k) {
       return (k == 0) ? 1 : (n * choose(n-1,k-1)) / k;
     }
     /**
     * Prints Pascal's triangle from top row 0 to row m
     */
    public void triangle(int m) {
      int n,k;
      for (n=0;n<=m;n++) {
        for (k=0;k<=n;k++)
          System.out.print("("+n+","+k+")="+choose(n,k));
        System.out.println();
      }
    }
}
```

1.2.4 Prime Factorization

The prime factorization, in increasing order, is known to be unique. Program 1.12 is a recursive algorithm, which outputs all prime factors, one after the other.

Program 1.12: Prime factorization.

```
public class PrimeFactor
{
   /** Recursive function that finds prime factor i and continues with n/i */
   public void factor(int n) { // prime factorization of n
     int i=2; while((i<n) && (n%i != 0)) i++; // find smallest factor
     System.out.println(i); // print factor
     if (i < n) factor(n/i); // recursive call with reduced term
   }
}
```

1.2.5 Gaussian Elimination

Assuming a unique solution, the Gaussian elimination algorithm solves linear equations of the form $MS = V$.

Program 1.13: Solving linear equations.

```
public class Gauss
{
    static final int DIM = 4;
    double M[][] = {{1.0,2.0,-1.0,-2.0},{1.0,3.0,-1.0,-2.0},{2.0,1.0,1.0,1.0},{3.0,1.0,2.0,1.0}};
    double V[] = {-6.0,-4.0,11.0,15.0};
    double S[] = new double[DIM];
    /**
     * Constructor for objects of class Gauss
     */
    public Gauss() {
      if (solve(M, V)) {
        for (int k = DIM-1; k>=0; k--) {
          S[k] = V[k];
          for (int i=(k+1); i<DIM; i++) S[k] -= (M[k][i]*S[i]);
          S[k] = S[k] / M[k][k];
        }
        System.out.println("Solution:");
        for (int i=0; i<DIM; i++)
          System.out.print(" " + S[i]);
        System.out.println();
      }
    }
    /**
     * Gauss method to compute S in M S = V, flag denotes full rank
     */
    public boolean solve(double M[][], double V[]) {
      for (int k=0; k<DIM-1; k++) {
        double max = M[k][k] > 0 ? M[k][k] : -M[k][k];
        int m = k;
        for (int i=k+1; i<DIM; i++) {
          double abs = M[i][k] > 0 ? M[i][k] : -M[i][k];
          if (max < abs) { max = M[i][k]; m = i; }
        }
        if (m != k) {
          for (int i=k; i<DIM; i++) { double t = M[k][i]; M[k][i] = M[m][i]; M[m][i] = t; }
          double t = V[k]; V[k] = V[m]; V[m] = t;
        }
        if(M[k][k] == 0) return false;
        for (int j=(k+1); j<DIM; j++) {
          double f = - M[j][k] / M[k][k];
          for (int i=k; i<DIM; i++) M[j][i] = M[j][i] + f*M[k][i];
          V[j] = V[j] + f*V[k];
        }
      }
      return true;
    }
}
```

Program 1.13 transforms the input matrix with the solution being maintained in the last column.

1.2.6 Min-Max Problem

Finding both the minimum and maximum in an array of size n is solved by 1) halving it, 2) finding the minima and maxima recursively, and 3) merging the results. Program 1.14 assumes that n is a power of 2.

Program 1.14: Concurrent computation of the minimum and the maximum.

```
public class MinMax
{
    int [] a;
    public class MM { int min; int max; };
    /**
     * Constructor for objects of class MinMax, N is number of elements
     */
    public MinMax(int N) {
      Random r = new Random();
      a = new int[N];
      for (int i=0;i<N;i++) {
        a[i] = r.nextInt(1000);
        System.out.print(a[i]+ ",");
      }
      System.out.println();
    }
    /**
     * search for minimum and maximum concurrently in interval x,y
     */
    MM search(int x, int y) {
      MM m = new MM();
      if (y <= x+1) { m.min = a[x]<a[y] ? x : y; m.max = a[x]<a[y] ? y : x; }
      else { // y > x+1, recursive call required
        MM l = search(x,(x+y)/2), r = search((x+y)/2+1,y);
        m.min = a[l.min] < a[r.min] ? l.min : r.min;
        m.max = a[l.max] > a[r.max] ? l.max : r.max;
      }
      return m;
    }
}
```

1.2.7 Quickselect and Quicksort

The *partition* of an array of length n along pivot p is a split into two parts, one populated with elements not smaller than, and one with elements not larger than p. The *selection problem* queries the k-th smallest element (for the *median* $k = \lfloor n/2 \rfloor$). After partitioning, *Quickselect* in Program 1.15 computes an answer recursively. *Quicksort* has the same partitioning step but continues in both array parts to eventually sort the sequence.

Program 1.15: Selection with *Quickselect* and sorting with *Quicksort*.

```
public class Quick
{
     private int a[];
    /**
     * Constructor: n array elements, k-th element selection
     */
    public Quick(int n, int k) {
      a = new int[n];
      Random r = new Random();
      for (int i=0;i<n;i++) a[i] = i;
      for (int i=1;i<n;i++) swap(i,r.nextInt(i));
      System.out.println("Quickselect "+ select(0,n-1,k));
      for (int i=1;i<n;i++) swap(i,r.nextInt(i));
      sort(0,n-1);
      for (int i=0;i<n;i++) System.out.print(a[i]+" ");
      System.out.println();
    }
    void swap(int i, int j) { int t = a[i]; a[i] = a[j]; a[j] = t; }
    /**
     * selects the kth element in interval left,right
     */
    public int select(int left, int right, int k) {
      int i,j,v;
      i = left; j = right+1; v=a[left]; // set pivot
      do { // partition wrt. pivot
        do j--; while (j>=i && v < a[j]);
        do i++; while (i<=j && a[i] < v);
        if (j > i) swap(i,j);
      } while(j >= i);
      swap(left,j); // move pivot, end of partitioning
      if (k == j) return a[k]; // element found
      return k < j ? select(left,j,k) : select(j+1,right,k);
    }
     /**
     * sorts interval left,right according to Hoares Quicksort algorithm
     */
    public void sort(int left, int right) {
      int i,j,v; // two indices and one temporary
      if (right-left > 0) { // interval is non-trivial
        i = left; j = right+1; v=a[left]; // set pivot
        do { // partition wrt. pivot
          do j--; while (j>=i && v < a[j]);
          do i++; while (i<=j && a[i] < v);
          if (j > i) swap(i,j);
        } while(j >= i);
        swap(left,j); // move pivot, end of partitioning
        if (j-left < right-i+1) { sort(left,j-1); sort(j+1,right); }
        else { sort(j+1,right); sort(left,j-1); }
      }
    }
}
```

1.3 Backtracking

Backtracking is a general problem set-and-reset solution technique.

1.3.1 Post's Correspondence Problem

Post's Correspondence Problem (PCP) is an *undecidable* problem as hard as the *halting problem*. The input of the problem consists of two finite arrays $x_1,\ldots,x_n$ and $y_1,\ldots,y_n$ of words over some alphabet Σ. A solution is a sequence of indices $(i_k)_{1\le k\le l}$ with $l \ge 1$ and $1 \le i_k \le n$ for all k, such that $x_{i_1}\ldots x_{i_l} = y_{i_1}\ldots y_{i_l}$. The problem is to decide whether or not such a solution exists.

Program 1.16: PCP.

```
public class PCP
{
   static final int DEPTH = 66;
   private int [] sol = new int[DEPTH];
   private String x[] = {"001","01","01","10"}; // initialise instance variables
   private String y[] = {"0","011","101","001"};
    /**
     * Constructor for objects of class PCP
     */
    public PCP() {
       backtrack(new String(""),new String(""),0);
    }
    /**
     * Backtracking method for pcpx, pcpy, truncated by depth
     */
    public void search(String pcpx, String pcpy, int depth) {
      if (depth == DEPTH) {
        if (pcpx.equals(pcpy)) {
          System.out.println(pcpx); System.out.println(pcpy);
          for (int j=0;j<DEPTH;j++) System.out.print(sol[j] +",");
          System.out.println(); System.exit(1);
        }
        else return;
      }
      for (int j=0;j<x.length;j++) {
        String tmpx = pcpx, tmpy = pcpy;
        pcpx += x[j]; pcpy += y[j];
        if (pcpx.startsWith(pcpy) || pcpy.startsWith(pcpx)) {
          sol[depth] = j; search(pcpx,pcpy,depth+1);
        }
        pcpx = tmpx; pcpy = tmpy;
      }
   }
}
```

Program 1.16 illustrates that undecidability does not necessarily mean that there is no algorithm that can solve instances to the problem. If the program runs, however, we cannot decide whether it will come up with a solution eventually, or if the problem is unsolvable.

1.3.2 Towers-of-Hanoi

The (3-peg) Towers-of-Hanoi problem asks for a rearrangement of n pairwise differently sized discs from peg a via peg b to peg c with the additional constraint that a larger disc must not be placed on a smaller one. Program 1.17 solves the problem of size n in exactly $2^n - 1$ moves.

Program 1.17: Towers-of-Hanoi.

```
public class Hanoi
{
    /**
     * Constructor for objects of class Hanoi
     */
    public Hanoi() {
      move(4,'a','b','c'); // call function
    }
    /**
     * moves of a stack in TOH problem
    */
     public void move(int n, char a, char b, char c) { // move stack
       if (n == 0) return; // stack is empty
       move(n-1,a,c,b); // recursive call, a to b via c
       System.out.println("move "+n+" from "+a+" to "+c);
       move(n-1,b,a,c); // recursive call, b to c via a
    }
}
```

1.3.3 Mazes

To find a way from start to goal in a maze is not immediate. Program 1.18 provides a computer solution in a random 2D grid with a 20% probability of an obstacle cell. The depth-first search algorithm looks in all four directions and marks every cell visited to avoid recomputation until the goal is eventually reached. The solution sequence is stored on a stack and printed in case of search success.

Program 1.18: A depth-first solver for a maze.

```
import java.util.Random;
public class Maze
{
    int maze[][], visited[][]; // maze and array to avoid search duplicates
    int goalx, goaly;
    /**
     * Constructor for (x,y)-sized Maze
     */
    public Maze(int x, int y) {
       Random r = new Random();
       maze = new int[x][y];
       visited = new int[x][y];
       for(int i=0;i<x;i++)
         for(int j=0;j<y;j++)
           maze[i][j] = visited[i][j] = 0;
       for(int i=1;i<x-1;i++)
         for(int j=1;j<y-1;j++)
     if (r.nextInt() % 5 != 0) maze[i][j] = 1;
       goalx = x-1; goaly = y-1;
       maze[goalx][goaly] = 0;
    }
    /**
     * An example of depth-first search at x,y trough maze
     */
    public void dfs(int x, int y) {
       if (maze[x][y] == 1 || visited[x][y] != 0) return; // failure
       System.out.println(" @ ("+x+","+y+")"); // output
       if (x == goalx && y == goaly) { // goal found
         System.out.println("goal found"); System.exit(1);
       }
       visited[x][y] = 1; // memorize, not to be visited again
       dfs(x,y+1); dfs(x,y-1); dfs(x+1,y); dfs(x-1,y); // calls to neighbors
    }
}
```

1.3.4 The Queens Problem

The Queens problem asks for an arrangement of n queens on an $n \times n$ chessboard so that no queen attacks any other. As search-free algorithms for one solution are known, Program 1.19 generates all possible oness.

An example output is

```
..@.
@...
...@
.@..

.@..
...@
@...
..@.
```

A related problem is placing a maximal number of knights on a chessboard. It is simple to see that on a standard board 32 is both the upper and lower bound.

Program 1.19: A solution to the queen's problem.

```
public class Queens
{
   private int[] a;
   /**
     * Constructor for n-Queens problem
     */
   public Queens(int n) {
     a = new int [n];
     solve(n,0);
   }
   /**
     * Check whether queen at q can be placed in row
     */
    public boolean legal(int q, int row) {
      for(int column=0;column<row;column++)
       if (a[column] == q || a[column]-q == column-row || q-a[column] == column-row)
         return false;
      return true;
    }
    /**
     * Solve all n-queens problem recursively for row j
     */
    public void solve(int n, int j) {
      if (j==n) {
        for (int i=0;i<n;i++) { // for each row
          for (int k=0;k<n;k++) // for each column
            if (i == a[k]) System.out.print("@"); else System.out.print(".");
          System.out.println();
        }
      }
      for (int q=0; q<n; q++)
        if (legal(q,j)) { a[j] = q; solve(n,j+1); }
    }
}
```

1.3.5 Sudoku

A *Sudoku* of size 9×9 has 3×3 blocks of 3×3 cells. All rows, columns and blocks must contain the numbers from 1 to 9 exactly once. An example of the in- and output behavior of Program 1.20 is

```
.........          987654321
.....3.85          246173985
..1.2....          351928746
...5.7...          128537694
..4...1..    ->    634892157
.9.......          795461832
5......73          519286473
..2.1....          472319568
....4...9          863745219
```

Program 1.20: Sudoku problem solver.

```
public class Sudoku
{
    public char s[] = {'0','0','0','0','0','0','0','0','0',
                       '0','0','0','0','0','3','0','8','5',
                       '0','0','1','0','2','0','0','0','0',
                       '0','0','0','5','0','7','0','0','0',
                       '0','0','4','0','0','0','1','0','0',
                       '0','9','0','0','0','0','0','0','0',
                       '5','0','0','0','0','0','0','7','3',
                       '0','0','2','0','1','0','0','0','0',
                       '0','0','0','0','4','0','0','0','9'};

    /**
     * calling the solver
     */
    public void solve() { show(); search(0); show(); }
    private void show() {
      for (int i=0;i<9;i++) {
        for(int j=0;j<9;j++)
         System.out.print(s[i*9+j] == '0' ? "." : s[i*9+j]);
        System.out.println();
      }
      System.out.println();
    }
    public boolean test(char d, int r, int c) {
      for(int i=0;i<9;i++)
         if (s[9*r+i] == d || s[9*i+c] == d || s[9*(r/3*3+i/3)+(c/3*3+i%3)] == d) return false;
      return true;
    }
    /**
     * backtrack search for a Sudoku solution at position pos
     */
    public boolean search(int pos) {
      if (pos == 81) return true;
      if (s[pos] > '0') { if (search(pos+1)) return true; }
      else
        for (int i=0;i<9;i++)
          if (test((char) ('1'+i),pos/9,pos%9)) {
            s[pos] = (char) ('1'+i); if (search(pos+1)) return true; s[pos] = 0;
          }
      return false;
    }
}
```

1.4 Heuristic Search

The process of problem solving can often be modeled as a search in a state space starting from some given initial state with rules describing how to transform one state into another. These rules are applied repeatedly to eventually satisfy a given goal condition.

Heuristic search algorithms execute a guided exploration in a space of states. While in computer science the term *heuristic* generally refers to rules of thumb, we use it as a lower bound that guides the solution search process.

1.4.1 Number Partitioning

In the *number partitioning problem* we are given a set $N = \{0, \ldots, n-1\}$ of objects with sizes $a_0, \ldots, a_{n-1}$; the task is to find a partition of N into $S \subseteq N$ and $N \setminus S$ with $\sum_{i \in S} a_i = \sum_{j \in N \setminus S} a_j$. If a_i with $i \in N$ are integers and the sum is odd, the problem is clearly unsolvable. The problem for sums that are even, however, is NP-hard, so that no efficient (polynomial-time) solver is known.

Program 1.21: Greedy and complete solution.

```
import java.util.Arrays;
import java.util.Random;
public class Partition
{
  int [] a;
  int max;
  public Partition(int n) {
    max = Integer.MAX_VALUE;
    Random r = new Random();
    a = new int[n];
    for(int i=0;i<n;i++) a[i] = r.nextInt(20);
    Arrays.sort(a); // increasing -> decreasing
    for(int i=0,j=n-1; i < (n/2); i++, j--) { int t = a[i]; a[i] = a[j];a[j] = t; }
    for (int i=0; i<n; i++) System.out.print(a[i]+",");
    completeGreedy(0,0);
    System.out.println("remaining difference is "+max);
  }
  public int greedy() {
    int diff = 0;
    for (int i=0;i<a.length-1;i++)
      if (diff < a[i]) diff = a[i] - diff; else diff = diff - a[i];
    return diff;
  }
  public void completeGreedy(int i, int diff) {
    System.out.println(i+" ["+diff+"]");
    if (i == a.length) { if (diff < max) max = diff; return; }
    completeGreedy(i+1,diff < a[i] ? (a[i] - diff) : (diff - a[i]));
    completeGreedy(i+1,diff+a[i]);
  }
}
```

The *greedy* solution in Program 1.21 places the next object on the respective smaller side and computes the difference in the sum of both sides. It presorts the elements in decreasing order. The solution extends to a complete search by allowing elements also to be placed on the other side.

1.4.2 The 15-Puzzle

The 15-Puzzle (see Figure 1.1) is a Childrens' toy. By sliding the tiles, a goal configuration (e.g, $0, \ldots, 15$) has to be found. Finding a solution in the optimal number of moves is hard. A lower bound is the Manhattan-Distance, i.e., the sum of horizontal and vertical distances of the tiles to their respective goal location. The backtrack solver in Program 1.22 includes this heuristic and is inspired by Rich Korf's solution in C. It includes the generation of solvable instances. Further precomputation accelerates successor generation.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 1.1: Goal state in the *15-Puzzle*.

Program 1.22: A *15*-Puzzle Solver: constructor and random position.

```
public class FifteenPuzzle
{
    public static final int X = 4, SIZE = 16;
    int [] s;
    public class Operators {
       int num;
       int [] pos = new int[4];
    }
    Operators [] oprs = new Operators[SIZE];
    int [][][] inc = new int [SIZE][SIZE][SIZE];
    int thresh;
    long generated, total;
    /**
     * Constructor for objects of class FifteenPuzzle
     */
    public FifteenPuzzle() {
      s = new int[SIZE];
      for (int i=0;i<SIZE;i++) oprs[i] = new Operators();
      for (int blank = 0; blank < SIZE; blank++) {
        oprs[blank].num = 0;
        if (blank > X - 1) oprs[blank].pos[oprs[blank].num++] = blank - X;
        if (blank % X > 0) oprs[blank].pos[oprs[blank].num++] = blank - 1;
        if (blank % X < X - 1) oprs[blank].pos[oprs[blank].num++] = blank + 1;
        if (blank < SIZE - X) oprs[blank].pos[oprs[blank].num++] = blank + X;
      }
      for (int tile = 1; tile < SIZE; tile++)
        for (int source = 0; source < SIZE; source++)
         for (int destindex = 0; destindex < oprs[source].num; destindex++) {
           int dest = oprs[source].pos[destindex];
           inc[tile][source][dest] =
             Math.abs((tile % X) - (dest % X)) - Math.abs((tile % X) - (source % X)) +
             Math.abs((tile / X) - (dest / X)) - Math.abs((tile / X) - (source / X));
        }
    }
    /**
     * Procedure to generate random (solvable) state, returns blank position
     */
    public int generate () {
      int blank, swaps;
      Random r = new Random();
      do {
        for (int index = 0; index < SIZE; index++) s[index] = index;
        blank = swaps = 0;
        for (int index = SIZE-1; index > 0; index--) {
          int other = r.nextInt(index + 1);
          if (other != index) {
            int temp = s[index]; s[index] = s[other]; s[other] = temp;
            swaps++;
          }
          if (s[index] == 0) blank = index;
          if (s[other] == 0) blank = other;
        }
        swaps = (swaps + blank % X + blank / X) % 2;
      } while (swaps == 1);
      return blank;
    }
```

Program 1.22 shows the search procedure including the loop that generates different instances. The statistics and the optimal solution length are printed.

Program 1.23: A *15*-Puzzle, solved with iterative-deepening search.

```
  /**
   * searches via depth-first iterative-deepening, cutting off when the depth plus the heuristic
       evaluation exceeds threshold;
   * blank current blank position, oldblank preceding blank position
   * g search depth, h heuristic estimate to goal
   * returns 1 and records the sequence of tiles moved in the solution, 0 otherwise
   */
  public int search (int blank, int oldblank, int g,int h) {
    int newblank;
    for (int index = 0; index < oprs[blank].num; index++)
      if ((newblank = oprs[blank].pos[index]) != oldblank){
        int tile = s[newblank];
        int newh = h + inc[tile][newblank][blank];
        generated++;
        if (newh+g+1 <= thresh) {
          s[blank] = tile; s[newblank] = 0;
          if ((newh == 0) || (search(newblank, blank, g+1, newh) == 1))
            return 1;
          s[newblank] = tile; s[blank] = 0;
        }
      }
    return 0;
  }
 /**
   * drives depth-first iterative-deepening, generates problems and solves it.
   */
 public void driver() {
  for (int problem = 1; problem <= 5; problem++) {
    int blank = generate();
    int initeval = 0;
    for (int pos = 0; pos < SIZE; pos++)
      if (s[pos] != 0)
        initeval += Math.abs((pos % X) - (s[pos] % X)) + Math.abs((pos / X) - (s[pos] / X));
    thresh = initeval;
    total = 0;
    int success = 0;
    do {
      generated = 0;
      success = search(blank, -1, 0, initeval);
      total = total + generated;
      thresh += 2;
      System.out.println("..... Problem "+ problem +" cost "+ (thresh-2) + " nodes generated "+ total);
    } while (success == 0);
    System.out.println("Solved Problem "+ problem +" cost "+ (thresh-2) + " nodes generated "+ total);
  }
 }
}
```

1.4.3 Ranking and Unranking

Ranking is a bijective mapping of a permutation of size n to a number in $\{0,\ldots,n!-1\}$. The fast ranking and unranking function in Program 1.24 goes back to Myrvold and Ruskey, based on the observation that a random permutation can be uniformly generated if for all k in $n-1,\ldots,1$ the values $\pi[k]$ are swapped with $\pi[rand(k)]$, where $rand(k)$ denotes a random number in $\{0,\ldots,k\}$.

Program 1.24: Ranking and unranking in linear time.

```
public class Rank
{
  static final int N = 9;
  int [] pi = new int[N];
  int [] inversepi = new int[N];
  private void unrank(int n, int r, int pi[]) {
    if (n>0) {
      int tmp = pi[n-1]; pi[n-1] = pi[r%n]; pi[r%n] = tmp;
      unrank(n-1,r/n,pi);
    }
  }
  private int rank(int n, int pi[], int inversepi[]) {
    if (n==1) return 0;
    int s = pi[n-1];
    int tmp = pi[n-1]; pi[n-1] = pi[inversepi[n-1]]; pi[inversepi[n-1]] = tmp;
    tmp = inversepi[s]; inversepi[s] = inversepi[n-1]; inversepi[n-1] = tmp;
    return s + n*rank(n-1,pi,inversepi);
  }
  public void test(int value) {
    pi[0] = 5; pi[1] = 8; pi[2] = 3; pi[3] = 6; pi[4] = 1; pi[5] = 0; pi[6] = 2; pi[7] = 4; pi[8] =
        7;
    for (int i=0;i<N;i++) inversepi[pi[i]] = i;
    System.out.println("rank value of permutation is "+rank(N,pi,inversepi));
    for (int i=0;i<N;i++) pi[i] = i;
    unrank(N,value,pi);
    for (int i=0;i<N;i++) System.out.print(pi[i]+", "); System.out.println();
  }
}
```

1.4.4 Peg Solitaire

Peg solitaire (see Figure 12.4) is a well-known single-player game, in which iterated jumps of pegs are followed by the extraction of the pegs jumped over. The best is to reduce the number of pegs down to one, preferably placed in the middle of the board. By its smaller runtime, Program 1.25 has been limited to a slightly easier instance than the common Greek cross.

Program 1.25: Solving Peg Solitaire.

```
public class Peg
{
    final int PEGS = 25;
    final int XDIM = 7;
    final int YDIM = 6;
    private int [] solx = new int [PEGS];
    private int [] soly = new int [PEGS];
    private char B[][] = {
        {'-','-','x','x','-','-'},
        {'-','-','x','x','-','-'},
        {'x','x','x','x','x','x'},
        {'x','x','o','x','x','x'},
        {'x','x','x','x','x','x'},
        {'-','-','x','x','-','-'},
        {'-','-','x','x','-','-'} };
    /**
     * Constructor for objects of class Peg
     */
    public Peg() {
        solve(PEGS);
    }
    /**
     * Solution method with number of remaining pegs
     */
    public void solve(int pegs) {
      if (pegs == 1) {
        for (int i=PEGS-1;i>0;i--)
          System.out.print("("+solx[i]+","+soly[i]+")");
        System.exit(1);
      }
      for (int i=0;i<XDIM;i++)
        for (int j=2;j<YDIM;j++)
          if (B[i][j] == 'o' && B[i][j-1] == 'x' && B[i][j-2] == 'x') {
          B[i][j-2] = B[i][j-1] = 'o'; B[i][j] = 'x';
          solx[pegs-1] = i; soly[pegs-1] = j;
          solve (pegs-1);
          B[i][j-2] = B[i][j-1] = 'x'; B[i][j] = 'o';
          }
      for (int i=0;i<XDIM;i++)
        for (int j=0;j<YDIM-2;j++)
          if (B[i][j] == 'o' && B[i][j+1] == 'x' && B[i][j+2] == 'x') {
          B[i][j+2] = B[i][j+1] = 'o'; B[i][j] = 'x';
          solx[pegs-1] = i; soly[pegs-1] = j;
          solve (pegs-1);
          B[i][j+2] = B[i][j+1] = 'x'; B[i][j] = 'o';
          }
      for (int i=2;i<XDIM;i++)
        for (int j=0;j<YDIM;j++)
          if (B[i][j] == 'o' && B[i-1][j] == 'x' && B[i-2][j] == 'x') {
          B[i-2][j] = B[i-1][j] = 'o'; B[i][j] = 'x';
          solx[pegs-1] = i; soly[pegs-1] = j;
          solve (pegs-1);
          B[i-2][j] = B[i-1][j] = 'x'; B[i][j] = 'o';
          }
      for (int i=0;i<XDIM-2;i++)
        for (int j=0;j<YDIM;j++)
          if (B[i][j] == 'o' && B[i+1][j] == 'x' && B[i+2][j] == 'x') {
          B[i+2][j] = B[i+1][j] = 'o'; B[i][j] = 'x';
          solx[pegs-1] = i; soly[pegs-1] = j;
          solve (pegs-1);
          B[i+2][j] = B[i+1][j] = 'x'; B[i][j] = 'o';
          }
    }
}
```

1.4.5 Traveling Salesman Problem

In the *traveling salesman problem* (TSP) we must visit each of n cities on a shortest tour exacty once. The input is a distance matix with enties $d_{i,j}$, $0 \leq i,j \leq n-1$, and the output a permutation π of $0,\ldots,n-1$ so that $d_{n-1,0}+\sum_{i=0}^{n-1} d_{\pi_i,\pi_{i-1}}$ is minimal.

Program 1.26 solves the problem. It presorts the distances in array `far`. For a constant runtime at each search node, we used a computer word to memorize and word-level bit-operations to modify the cities visited.

Program 1.26: Initialization of the TSP solver.

```
import java.util.Random;

public class TSP
{
    public class State {
          int h, g, depth, city;
          long used;
          public State() { used = 0L; g = h = depth = city = 0; }
    }
    public class Memory {
          int top, max;
          State [] old;
          public Memory(int n) {
            max = n*n; top = 0; old = new State[max+1];
            for (int i=0;i<max+1;i++) old[i] = new State();
          }
    }
    static final int N = 50;
    int expansions;

    /**
     * Constructor for objects of class TSP, initializing distance matrix
     */
    public TSP() {
          start = 0;
          Random r = new Random(100);
          for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                dist[i][j] = (i == j) ? 0 : r.nextInt(100);
          for (int i = 0; i < N; i++) {
            newState[i] = new State();
            for (int j = 0; j < N; j++)
                far[i][j] = j;
            for (int j = 0; j < N-1; j++)
                for (int k = j+1; k < N; k++)
                  if (dist[i][far[i][j]] < dist[i][far[i][k]]) {
                      int temp = far[i][j]; far[i][j] = far[i][k]; far[i][k] = temp;
                  }
          }
    }
```

Program 1.27 lists the branch-and-bound search. If a better solution than the current best one is found, it sets variable α to this value. (Programs 23.1 and 23.2 show a solution of the assignment problem (AP) used as a heuristic employing the Hungarian algorithm.)

Program 1.27: Searching a solution for the TSP.

```
/**
 * Solving the TSP problem with heuristic h (h=0, h=1 Hungarian method)
 */
public int solve(int h) {
     expansions = 0;
     int alpha = Integer.MAX_VALUE;
     int top = stack.top++;
     stack.old[top].g = stack.old[top].depth = 0;
     stack.old[top].city = start;
     used = 0L;
     stack.old[top].h = heuristic(h,0,0,0);
     stack.old[top].used = used = (1L << start);
     while (stack.top != 0) {
       top = --stack.top;
       int depth = stack.old[top].depth;
       int city = stack.old[top].city;
       tour[depth] = city;
       if (depth == N - 1) {
            if (stack.old[top].g + dist[city][start] < alpha) {
              alpha = stack.old[top].g + dist[city][start];
              System.out.println("cost: " + alpha +"(" + expansions + ")");
            }
            continue;
       }
       used = stack.old[top].used;
       int cost = stack.old[top].g;
       int opindex = 0;
       expansions++;
       for(int i=0; i < N; i++) {
            if (((used >> far[city][i]) & 1L) > 0) continue;
            int newcity = far[city][i];
            newState[opindex].depth = depth+1;
            next[opindex] = opindex + 1;
            int g = cost + dist[city][newcity];
            newState[opindex].g = cost + dist[city][newcity];
            newState[opindex].city = newcity;
            used &= ~(1L << start);
            newState[opindex].h = heuristic(h,g,newcity,depth);
            used |= (1L << start);
            used |= (1L << newcity);
            newState[opindex].used = used;
            used &= ~(1L << newcity);
            opindex++;
       }
       next[opindex-1] = N;
       for(int i=0; i != N; i = next[i]) {
            if (newState[i].g + newState[i].h >= alpha) continue;
            int newtop = stack.top++;
            stack.old[newtop].city = newState[i].city;
            stack.old[newtop].used = newState[i].used;
            stack.old[newtop].g = newState[i].g;
            stack.old[newtop].h = newState[i].h;
            stack.old[newtop].depth = newState[i].depth;
       }
     }
     return alpha;
}
public int heuristic(int h, int g, int city, int depth) {
     return (h == 1) ? HungarianMethod(g,city,depth) : 0;
}
```

1.5 Randomization

Randomization plays an important role in computer science; randomized algorithms are often easier and more effective than deterministic ones. Frequently, they call the pseudo-random number generator of Lehmer. One distinguishes *Monte-Carlo* methods, being *mostly correct* from *Las-Vegas* methods, which are always correct, but have a varying running time.

1.5.1 Randomized Prime Number Tests

Large primes are important in public-key cryptographic systems like RSA. Miller (Selfridge) and Rabin used a random test for prime numbers that is shown in Program 1.29. The algorithm has been found independently by Solovoy and Strassen. It errs with some probability, but the error can be reduced to the level of hardware reliability by repeated invocations. The *Sieve of Eratostenes* is far less efficient, but deterministic. For a potential prime x for a growing $2 \leq n \leq \lfloor\sqrt{x}\rfloor$ we check if n divides x. In Program 1.28 we set $\sqrt{x}$ to 100 so that values $x < 10{,}000$ can be checked for primality.

Program 1.28: Finding prime numbers following Erastosthenes.

```
public class Eratosthenes
{
  public void isPrime(int x) {
    int i, j=0, n=3;
    int primes[] = new int[100];
    primes[0] = 2;
    boolean prime = true;
    if (x%2==0 && x!=2) { prime = false; n = 2; }
    while (n <= Math.sqrt(x)) {
      i = 0;
      boolean prime2 = true;
      while (primes[i]*primes[i]<=n) {
        if (n%primes[i]==0) { prime2 = false; break; }
        i++;
      }
      if (prime2) {
        if (x%n==0) { prime = false; break; }
        if (j<100) primes[j++] = n;
      }
      n+=2;
    }
    if (prime) System.out.println(x+" is a prime number!");
    else System.out.println(x+" = "+n+"*"+(x/n));
  }
}
```

Program 1.29: The random computation of prime numbers.

```
public class Prime
{
    /**
     * Probabilistic prime detection of n, with a random value in [0..n]
     */
    boolean witness(int a, int n) {
      int d = 1, k = log(n)+1;
      for (int i=k-1; i>=0; i--) {
        d = (d*d) % n;
        if ((((n-1) >> i) & 1) == 1) d = (d*a) % n;
      }
      return (d == 1);
    }
    public boolean test(int n) {
      Random r = new Random();
      for (int i = 0;i<100;i++) {
        int a = 1 + r.nextInt(n-1);
        if (!witness(a,n)) return false;
      }
      System.out.println("probably prime");
      return true;
    }
}
```

1.5.2 Mister X

Mister X selects a random number that has to be determined in as few queries as possible.

Program 1.30 comes with an automated computer player, which solves the problem in the optimal number of queries. It halves the interval in which the random number is located, and, thus, applies *binary search*.

Program 1.30: The game *Mister X*.

```
public class MisterX
{
   private int x, left, right, trials;
    /**
     * Constructor for objects of class MisterX
     */
    public MisterX(int d) {
      trials = left = 0; right = d;
      Random r = new Random();
      x = r.nextInt(d);
      System.out.println("Mister X has chosen a number between 0 and " + (d-1));
    }
     /**
     * Optimal play against Mister X via binary search
     */
    public void solve() {
        trials = 0;
        while (true) {
          trials++;
          int y = (left+right)/2;
          System.out.println("Mister X – my " + trials + ". guess is " + y);
          if (x<y) { System.out.println("...too big"); right = y; }
          if (x>y) { System.out.println("...too small"); left = y; }
          if (x==y) { System.out.println("...correct, "+ trials + " trials"); break; }
        }
    }
    /**
     * Guessing z against Mister X
     */
    public void guess (int y) {
      trials++;
      System.out.println("Mister X – your " + trials + ". guess is " + y);
      if (x<y) System.out.println("...too big");
      if (x>y) System.out.println("...too small");
      if (x==y) System.out.println("...correct, "+ trials + " trials");
    }
 }
```

1.5.3 Mastermind

Mastermind is a game in which a random code has to be broken. Statistics about the proper choice of a digit and the correct location are made. Donald E. Knuth showed that it is possible to find a code of length 4 as a selection of six numbers in at most five queries.

Program 1.31 implements the game, in which the computer chooses the random code in its constructor. All n input numbers are pairwise different, i.e., for $a_0, \ldots, a_{n-1}$ we have $a_i \neq a_j$ for all $0 \leq i \neq j < n$.

Program 1.31: The game *Mastermind*.

```
public class MasterMind
{
    private int [] secret;
    private int [] input;
    int size, trials;
    /**
     * Constructor for objects of class MasterMind, parameter n<9
     */
    public MasterMind(int n) {
      size = n; trials = 0;
      Random r = new Random();
      secret = new int[size]; input = new int[size];
      for(int i=0;i<size;i++) {
        boolean duplicate = false;
        do {
          duplicate = false;
          secret[i] = 1+r.nextInt(9);
          for(int j=0;j<i;j++)
          if (secret[i] == secret[j]) duplicate = true;
        } while (duplicate);
      }
      System.out.println("Mastermind has chosen " + size + " distinct numbers between 1 and 9");
    }
    /**
     * Guessing a code with guess y an n-th digit number
     */
    public void guess(int y) {
      trials++;
      System.out.println("your "+ trials + ". number is " + y);
      boolean hurray = true;
      for(int i=0;i<size;i++) { input[size-i-1] = y % 10; y = y / 10; }
      for(int i=0;i<size;i++) {
        boolean red = false;
        boolean black = false;
        for(int j=0;j<size;j++)
           if (input[i] == secret[j]) red = true;
         if (input[i] == secret[i]) {
           System.out.println((i+1) + ". position, MM code "+secret[i]+" = your code " + input[i]);
         black = true;
         }
         else{
           if (red) System.out.println((i+1) + ". position, MM _ ~ your code " + input[i]);
         else System.out.println((i+1) + ". position, MM _ # your code " + input[i]);
         }
         hurray = hurray & black;
       }
      if (hurray) System.out.println("hurray, you broke the code");
    }
}
```

1.5.4 Nim

Nim is a two-player turn-taking game, in which for every move a number of sticks must be removed from a chosen row. The player who removes the last sticks wins the game. There is a known winning strategy imple-

mented in Program 1.32: if and only if the bitwise xor of the row stick count is zero, then the position is lost for the player to move.

Program 1.32: The game *Nim*.

```
public class Nim
{
    private int a[];
    private int size;
    /**
     * Constructor for objects of class Nim
     */
    public Nim(int n)
    {
        a = new int [n];
        size = 0;
        Random r = new Random();
        while (size < n)
            a[size++] = r.nextInt(10);
    }
    public void take(int i, int j) {
        a[i] -= j;
    }
    public int xor () {
        int r = 0;
        for(int i=0;i<size;i++)
            r = r ^ a[i];
        return r;
    }
    public void solve() {
        for (int i=0;i<size;i++) {
            for (int j=1;j<=a[i];j++) {
                if (a[i] >= j) {
                    int temp = a[i];
                    a[i] -= j;
                    if (xor() == 0) {
                        System.out.println("Taking " + j + " sticks from row " + i);
                        print();
                        return;
                    }
                    a[i] = temp;
                }
            }
        }
    }
}
```

1.5.5 Snake

Snake is a one-player challenge. The head moves according to the given direction in a maze while the body extends until the snake bites itself. Program 1.33 shows a prototypical implementation.

Program 1.33: The game *Snake*.

```
public class Snake
{
   int maze[][]; // maze with walls that block
   int startx, starty, score, length; Random r;
   public Snake(int x, int y) {
      r = new Random(); maze = new int[x][y]; score = length = 0;
      for(int i=1;i<x-1;i++)
        for(int j=1;j<y-1;j++)
          if (r.nextInt() % 30 == 0) maze[i][j] = 1;
      for(int i=0;i<x;i++) maze[i][0] = maze[i][y-1] = 1;
      for(int i=0;i<y;i++) maze[0][i] = maze[x-1][i] = 1;
      startx = x-2; starty = y-2; maze[startx][starty] = 0;
      while (true) {
        if (maze[startx-1][starty] != 0 && maze[startx+1][starty] != 0 &&
            maze[startx][starty-1] != 0 && maze[startx][starty+1] != 0) {
          System.out.println("No further progress."); System.exit(1); }
        int l = r.nextInt() % 4;
        if (l == 0 && maze[startx-1][starty] == 0) up();
        else if (l == 1 && maze[startx][starty-1] == 0) left();
        else if (l == 2 && maze[startx+1][starty] == 0) down();
        else if (l == 3 && maze[startx][starty+1] == 0) right();
      }
   }
   public void print() {
     try{ Thread.sleep(500); } catch(InterruptedException e){}
     for(int i=0;i<maze.length;i++) {
         for(int j=0;j<maze[i].length;j++) {
          if (maze[i][j] == 1) System.out.print("X");
          else if (i == startx && j == starty) System.out.print("@");
          else if (maze[i][j] < 0) { System.out.print("o"); maze[i][j]++; }
          else System.out.print(".");
         }
        System.out.println();
     }
     length = ++score / 10;
     System.out.println("Score: " + score + ". Length = " + length + ".");
   }
   public void left() {
     if (maze[startx][starty-1] != 1) maze[startx][--starty] = -length; print(); }
   public void down() {
     if (maze[startx+1][starty] != 1) maze[++startx][starty] = -length; print(); }
   public void up() {
     if (maze[startx-1][starty] != 1) maze[--startx][starty] = -length; print(); }
   public void right() {
     if (maze[startx][starty+1] != 1) maze[startx][++starty] = -length; print(); }
}
```

1.5.6 PacMan

PacMan is a well-known Arcade game. A simplified version is played in a maze (see Program 1.34). Ghosts are numbered. The game ends if one ghost hits the player.

Program 1.34: The game *PacMan*.

```
public class PacMan
{
    int maze[][]; // maze with walls that block
    int visited[][]; // memorization to avoid duplicates
    int startx, starty, m, p;
    int goalx[], goaly[];
    Random r;
    public PacMan(int x, int y) {
       m = 4; p = 0;
       maze = new int[x][y]; visited = new int[x][y]; goalx = new int[m]; goaly = new int[m];
       r = new Random();
       for(int i=1;i<x-1;i++)
         for(int j=1;j<y-1;j++)
           if (r.nextInt() % 5 == 0) maze[i][j] = 1;
       for(int i=0;i<x;i++) maze[i][0] = maze[i][y-1] = 1;
       for(int i=0;i<y;i++) maze[0][i] = maze[x-1][i] = 1;
       for(int i=0;i<m;i++) { goalx[i] = 1; goaly[i] = 1; }
       startx = x-2; starty = y-2;
       p = pills(startx,starty);
       while (true) {
         if (r.nextInt() % 4 == 0) monster();
         int l = r.nextInt() % 4;
         if (maze[startx-1][starty] == 0) up();
         else if (maze[startx+1][starty] == 0) down();
         else if (maze[startx][starty-1] == 0) left();
         else if (maze[startx][starty+1] == 0) right();
         else if (l == 0) up(); else if (l == 1) down();
         else if (l == 2) left(); else if (l == 3) right();
       }
    }
    public void left() {
      if (maze[startx][starty-1] != 1) maze[startx][--starty] = -1; print(); }
    public void down() {
      if (maze[startx+1][starty] != 1) maze[++startx][starty] = -1; print(); }
    public void up() {
      if (maze[startx-1][starty] != 1) maze[--startx][starty] = -1; print(); }
    public void right() {
      if (maze[startx][starty+1] != 1) maze[startx][++starty] = -1; print(); }
    public void monster() {
      int oldx, oldy;
      for(int i=0;i<m;i++) {
        do {
          int l = r.nextInt() % 8;
          oldx = goalx[i]; oldy = goaly[i];
          if (l == 0 && goaly[i] > 0) goaly[i] = goaly[i]-1;
          else if (l == 1 && goalx[i] < maze.length-1) goalx[i] = goalx[i]+1;
          else if (l == 2 && goalx[i] > 0) goalx[i] = goalx[i]-1;
          else if (l == 3 && goaly[i] < maze[0].length-1) goaly[i] = goaly[i]+1;
          else if (l == 4 && startx < goalx[i]) goalx[i] = goalx[i]-1;
          else if (l == 5 && startx > goalx[i]) goalx[i] = goalx[i]+1;
          else if (l == 6 && starty < goaly[i]) goaly[i] = goaly[i]-1;
          else if (l == 7 && starty > goaly[i]) goaly[i] = goaly[i]+1;
          if (maze[goalx[i]][goaly[i]] == 1) { goalx[i] = oldx; goaly[i] = oldy; }
        } while (goalx[i] == oldx && goaly[i] == oldy);
        if (startx == goalx[i] && starty == goaly[i]) System.exit(1);
      }
    }
    public int pills(int x, int y) {
      if (maze[x][y] == 1 || visited[x][y] != 0) return 0;
      visited[x][y] = 1;
      return 1 + pills(x,y+1) + pills(x,y-1) + pills(x+1,y) + pills(x-1,y);
    }
}
```

1.6 Bibliographic Notes

The science of computing is a rather young research discipline founded by Alan Turing, Max Neumann, David Hilbert, Wilhelm Ackermann, Kurt Gödel, Alonzo Church, Stephen C. Kleen, and Emil Post, just to name a few. In his 1936 manifesto *On computable numbers, with an application to the Entscheidungsproblem* Turing introduces the notation of computability. Later on, in 1968, 1969, and 1973 Donald E. Knuth wrote *The Art of Computer Programming*, a foundation of modern computer science. The three volumes study Fundamental Algorithms (1), Semi-Numerical Algorithms (2), Searching and Sorting (3).

In the late 1970s the common basis to teach programming was *The C Programming Language* by Brian Kernighan and Dennis Ritchie (since 2012 there also is an electronic version). While being an outcome of a tutorial, the book is not a textbook for teaching. The proposed ANSI C Standard listed in Backus-Naur format, however, is still used in programming practice. In his monograph *Data Structures and Algorithms* Kurt Mehlhorn reflected the Status Quo in algorithm analysis. The work addresses three books: Sorting and Searching (1), Graph Algorithms and NP-Completeness (2), as well as Multi-Dimensional Search and Computational Geometry (3). In 1983, Robert Sedgewick published his book *Algorithms*, which was ported to Java and C++. It introduces sorting and searching, graphs, strings and further material. In 1990, the textbook *Introduction to Algorithms* of Thomas Cormen, Charles Leiserson and Ronald Rivest appeared as a compendium on algorithmics (and attracted Clifford Stein as a fourth author). Besides classic subjects like searching and sorting, advanced topics like Fibonacci heaps and the discrete Fourier transformation are covered, and an introduction to complexity theory is given.

Object-oriented programming was popularized in the mid 1980s; central was the work *The C++ Programming Language* by Bjarne Stroustrup. Even though more than one third of the book is a reference manual, its technical insights persist. The Massachusetts Institute of Technology (MIT) used the LISP-like programming language *Scheme* following the book *Structure and Interpretation of Computer Programs* by Julie Sussman, Harold Abelson and, later, Gerald Jay Sussmann for teaching algorithms. The first Turing-Award Winner Alan Jay Perlis (ALGOL) wrote the preface. Characters like Ben Bitdiddle, Eva Lu Ator, Louis Reasoner, Alyssa P. Hacker, Cy D. Fect and Lem E. Tweakit were introduced; similar to Andrew Tanenbaum's *ostrich* algorithm, which solves a problem merely by pretending that there is no problem at all. *Java* emerged as the programming language of choice, and applets connected programs with the Internet. Other languages like *Ruby*, an object-oriented scripting language, were designed together with *Rails* for the design of web applications. Agile software development and *extreme programming* led to programming models like *Scrum*.

Objects First with Java by David J. Barnes and Michael Kölling (Prentice Hall / Pearson Education, 2012) advertises a software technology paradigm that aims at substituting existing ones. It takes a graphical interface to start with and continues with the interaction of several objects kept in lists via interfaces. Next, there are design principles and typographic conventions, and methods for error handling. Abstract classes then lead to larger software projects. Linear lists and hash tables are used, but not explained, and there is almost no performance analysis of algorithms. In a similar threat, Becker and Buck et al. take *Karel the Robot* to introduce object orientation in a five-weeks course *Instantiating and Using Objects* (1), *Extending Existing Classes* (2), *Selection and Iteration* (3), *Methods with Parameter* (4), and *Instance Variables* (5).

Algorithm Engineering connects theory and practical hardware in the design of algorithms. Its core is a cycle driven by falsifiable hypotheses. It consists of design, analysis, implementation and experimental evaluation of practicable algorithms. Realistic models for both computers and applications, as well as algorithm libraries and collections of real input data allow a close coupling to applications. The textbook of Kurt Mehlhorn and Peter Sanders *Data Structures and Algorithms – The Basic Toolbox* reflects this trend.

Recommended references to randomized search are the books *Randomized Algorithms* by Motwani and Raghavan and *Probability and Computing*. by Mitzenmacher and Upfal. The book *Primality Testing in Polynomial Time* by Dietzfelbinger gives an insightful survey of the primes problem and introduces a recent deterministic algorithm that is polynomial.

Chapter 2
Shortest Paths

In state-space search the initial and the goal states are nodes in a graph. Edges between the nodes are labeled with cost. The task in single-source shortest path search is to find an optimal plan in the form of a sequence of edges that have the smallest cost total. In general graphs, the best-known algorithms are variants of Dijkstra's approach. Nonetheless, after more than 50 years there are still many interesting cases in which it can be improved significantly.

Starting from a single one, Dijkstra's method actually determines the shortest paths to *all* other nodes in a weighted graph, for example to generate pattern databases.

In this chapter we vary the underlying priority queue data structure and conduct experiments in commercial game maps with Euclidean edge weights. We compare general implementations of binary, pairing and Fibonacci heaps with bucket implementations, including radix heaps and bucket maps. As domain-specific highlights, we study theoretical properties of factorizing edge costs and exploiting cache efficiencies in the memory layout of the graph.

2.1 Introduction

For[1] optimally solving task planning problems, where actions are assigned to costs, the single-source shortest path (SSSP) algorithm of Dijkstra applied to a graph with weight (alias *cost* or *distance*) assigned to the edges is one apparent implementation option.

For a consistent heuristic, where $w(u,v) \geq h(u) - h(v)$ for all edges (u,v), the A* algorithm is equivalent to Dijkstra's method applied to a compiled graph. The heuristic h reweights all the edges $e = (u,v)$ to convert the original graph $G = (V,E,w)$ according to the formula $w'(u,v) = w(u,v) + h(v) - h(u)$ into $G = (V,E,w')$ together with an offset $h(s)$ assigned to the initial state s.

We solve the shortest path problem where the search does not terminate at a goal. In this setting, lower bound heuristics are not helpful, as they only modify the ordering of visiting nodes. Hence, algorithmic improvements to Dijkstra's original method are essential.

For precomputing heuristic estimates (alias pattern databases) the all-target single-source shortest path problem arises in the inverse of a state space graph abstraction, with a relaxation applied to the nodes and all directed edges reversed. The computed goal distance serves as a lower bound.

But what is the best way of implementing Dijkstra's algorithm? Of course, the answer largely depends on the graph itself, the weight function applied, and the level of engineering in the implementation. The question we

[1] This chapter is based on joint work with Asger Bruun, Jyrki Katajainen, and Jens Rasmussen. It puts together and improves the work from [88, 182].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_2

stress in this chapter is whether the general priority queues are competitive in rather regular environments found in AI, or if specialized implementations are superior.

We experiment with grid graphs that are common in game playing and robotics research. For online navigation, the performance for finding shortest paths determines the applicability of a method. We assume that the graphs to be searched are stored in (main) memory, with nodes to be addressed in constant time, and study maps of commercial games. We prefer non-unit edge costs such as imposed by the Euclidean distance metric in an eight-connected (octile) grid.

We introduce and analyze specialized data structures that exploit either

- the limited set of available cost values, or
- the 2D graph layout to prefer a cache-friendly exploration.

For the former we introduce a *factorized* algorithm that operates on a matrix of buckets, and for the latter we apply *flood-filling*, where a different traversal order to Dijkstra's algorithm leads to reopening of nodes. The hope is that the search becomes faster and compensates for this additional work.

2.2 Dijkstra's Algorithm

Finding shortest paths in a directed and weighted graph $G = (V, E, w)$ with $E \subseteq V \times V$, and $w : V \to \mathbb{R}^+$ is essential to many areas of computer science. While for general graphs (assuming an adjacency list representation) the input is of size $O(|E| + |V|)$, in grid graphs with a bounded branching factor at each cell, we have $|E| = O(|V|)$, so that the input (and the output) has size $O(|V|)$.

While it is still an open question if there is an algorithm for finding shortest paths in general graphs that is linear in the size of the graph, for special graph classes, like planar graphs and undirected graphs with integer weights, the problem has been theoretically solved. The solutions, however, are involved: for planar graphs, graph separators are recursively applied, while for undirected graphs with integer weights heaps that are efficient only for very large values of n have been proposed. Essentially, even for simpler types of graphs, the choice of a proper data structure remains challenging.

Program 2.1: Template for Dijkstra's algorithm.

```
template<typename H, typename N, typename K>
void dijkstra(vector<N*> vertices) {
  H pq;
  for (int i=0;i<vertices.size();i++) vertices[i]->visited = none;
  N* s = vertices[0];
  s->cost = 0; s->label = open;
  pq.insert(s);
  while (!pq.is_empty()) {
    N* t = pq.find_min(); t->label = closed;
    K d = t->cost;
    for (int i = 0; i<t->succs;i++) {
      N* u = vertices[t->successor[i]];
      K c = d + t->w[i];
      if (u->label != closed) {
        if (u->label == none) { u->cost = c; u->label = open; pq.insert(u); }
        else if (u->cost > c) pq.decrease_key(u, c);
      }
    }
    pq.extract_min(t);
  }
}
```

For general priority queues, in a precomputation stage, the grid is scanned once and compiled into a weighted graph with nodes for each cell and edges for each link between two adjacent cells.

One implementation uses a bitvector for tagging elements *open* (visited). While for dense graphs this option incurs an acceptable overhead, for sparse graphs with limited branching, finding the next open node can be costly.

As there are different expositions of Dijkstra's algorithms, Program 2.1 shows the engineered code with template types for priority queue (H), node (N) and key (K).

The refined implementation uses additional labels (*none*, *closed* or *open*) for nodes to avoid redundant work. It assumes a merge of graph and queue nodes (thus, the *joint node representation* includes the label *state*; a linked list of edges; an *element* for storing the travel distances, as well as pointers for linking the elements in the heap). An edge is a pair of a successor node ID and according weight. In some graphs such advanced node representation is more important than the proper choice of the data structure. One reason is avoiding fragmented memory allocation, another is dropping the efforts for preserving the bijection between graph and heap nodes.

The priority queue itself provides the usual operations *is-empty*, *insert*, *find-min*, *delete*, *decrease-key*, and *extract-min*.

2.3 General Priority Queues

With binary, Fibonacci and pairing heaps we look at general priority queues that can be used to find shortest paths in any directed graph with totally-ordered weight function. They are general in the sense that they are supporting the full set of priority queue operations, including *delete* and *decrease-key* (Fibonacci and pairing heaps also provide an efficient *meld* operation). This flexibility of a priority queue, however, comes at a price, as these two operations usually require maintaining *handles* to nodes, given that nodes are moving within the heap.

2.3.1 *k*-ary Heaps

The first structure we consider is a *binary heap*, as suggested by Floyd and Williams in the heapsort algorithm. (Chapter 3 provides a closer look at binary heaps.) As all operations are logarithmic, it is easy to show that in total we have $O((|E|+|V|)\lg|V|)$ worst-case time (with lg denoting $\log_2$) for finding shortest paths with a priority queue based on binary heaps; there are $|V|$ delete-min and $|E|$ decrease-key operations. We have implemented algorithmic enhancements like k-ary heaps that have smaller height and larger branching factor, but the improvements with respect to binary heaps were minor (slightly better results have been found with $k = 4$). For the sake of clarity, we stick to binary heaps and $k = 2$.

2.3.2 Fibonacci Heaps

Fibonacci heaps are doubly-linked root lists of heap-ordered trees. They have been characterized as a lazy-meld version of a binomial queue, which in turn, is a set of binomial trees.

A binomial tree B_n is a tree of height n with 2^n nodes in total (and i nodes at depth i) and consists of two trees B_{n-1}. Binomial queues are unions of heap-ordered binomial trees. Tree B_i is represented in queue Q if and only if the ith bit in the binary representation of n is set. Several trees of rank i may be represented in one Fibonacci heap. Consolidation traverses the linear list and merges trees of the same rank so that each rank becomes unique.

It is well known that the standard priority queue operations of extracting the minimum and decreasing the key of a node in a Fibonacci heap results in $O(|E|+|V|\lg|V|)$ worst-case time for SSSP. This efficiency is mainly due to the amortized constant time for *decrease-key*. There are priority queues with worst-case constant *decrease-key* operations and ones that provide a smaller number of comparisons, but for basic cost data types these structures were less performant. There are several Fibonacci heap variants; including *simple*, *eager*, and *lazy* ones (see Program 2.2 at the end of this chapter for one example).

2.3.3 Pairing Heaps

A pairing heap is a heap-ordered (not necessarily binary) self-adjusting tree. The basic operation is *pairing*, which combines two pairing heaps by attaching the root with the larger key to the other root as its leftmost child. For two pairing heaps with respective root values l_1 and l_2, *pairing* inserts the first as the leftmost subtree of the second if $l_1 > l_2$, and otherwise inserts the second into the first as its leftmost subtree.

Pairing takes constant time, and the minimum is found at the root. In a multiway tree representation realizing the priority queue operations is simple. Insertion pairs the new node with the root of the heap; *decrease-key* splits the node and its subtree from the heap (if the node is not the root), decreases the key, and then pairs it with the root of the heap; *delete* splits the node to be deleted and its subtree, performs a *delete-min* on the subtree, and pairs the resulting tree with the root of the heap. The *delete-min* operation removes and returns the root, and then, in pairs, pairs the remaining trees. Finally, the remaining trees from right to left are incrementally paired (see Program 2.3 at the end of this chapter).

Since the multiple-child tree representation is difficult to maintain, the child-sibling binary tree representation for pairing heaps is used, in which siblings are connected as follows. The left link of a node accesses its first child, and the right link of a node accesses its next sibling, so that the value of a node is less than or equal to all the values of nodes in its left subtree.

The analysis for pairing heaps, e.g., with two-phase root consolidation is involved. Nonetheless, since the empirical work of Stasko and Vitter there is strong evidence that pairing heaps perform well in practice.

2.4 Bucket Priority Queues

There are priority queue data structures that exploit the data type of the keys. While for general priority queues we only imposed a total order, for bucket-based structures we require integer keys with a maximum edge weight $C = \max_{e \in E} w(e)$. As we aim at general weight functions, we scaled and truncated real-valued distance values like $\sqrt{2}$. Due to this approximation of floating-point data considerably large integer edges costs are generated.

In an one-level bucket $C+1$ buckets suffice, but, frequently, a maximum-cost many buckets were used, as in Korf's A* implementation for the 15-Puzzle, code fragments of which are shown in Program 2.4.

Program 2.4: Code fragment for bucket-based A* search for 15-Puzzle.

```
while (table < FULL) {
  while (open[bestf] == MAX) bestf++;
  index = open[bestf];
  depth = expand(index);
  close(index);
}

int expand (int index) {
  int nps[WORDS];
  int newblank;
  int blank = unpack(old[index].ps, s);
  int newg = old[index].g+1;
  expanded++;
  for (int opindex = 0; opindex < oprs[blank].num; opindex++) {
    if ((newblank = oprs[blank].pos[opindex]) != old[index].oldblank) {
      int tile = s[blank] = s[newblank];
      s[newblank] = 0;
      int newh = old[index].h + manhat[tile][blank] - manhat[tile][newblank];
      if (newh == 0 && newg <= bestf) return (newg);
      pack(s, nps);
      int hashval = hash(nps);
      int newindex = search(nps, hashval);
      if (newindex == -1)
        insert(nps, hashval, blank, newg, newh);
      else if (old[newindex].g > newg) {
        close(newindex);
        reopened++;
        insert(nps, hashval, blank, newg, newh);
      }
      s[newblank] = tile;
      s[blank] = 0;
    }
  }
  return 0;
}
```

The worst-case time performance for an one-level bucket representation of the priority queue is $O(C \cdot |V| + |E|)$, as we might have $C-1$ empty buckets in between two non-empty ones. For a small constant value C, the shortest paths algorithm based on buckets has optimal time complexity, but for more complex cost values, the performance degrades quickly.

Most shortest-paths algorithms based on buckets assume a *monotone* cost function, that is the cost of each successor node is larger than the one of the current one. This leads to *addressable* priority queues that only need to support *push* (aka *insert*), *top* (aka *find-min*) and *pop* (aka *delete-min*).

The reason for buckets to work is that while a node might be reached more than once with different cost values, it will be expanded first with optimal cost. For octile grids of size $X \times Y$ this leads to the implementation shown in Program 2.5. As imposed by the benchmarks, *cutting corners* is not allowed. We maintain traversal information such as distance or visitedness in tables and assume access to the information if a cell is free or occupied. Pairs of distance and grid location are stored in the priority queue.

Program 2.5: Bucket-based shortest paths search.

```
void bucket_dijkstra(int startx, int starty) {
  heap pq;
  for (int i=0;i<X;i++)
    for (int j=0;j<Y;j++) { dist[i][j] = MAXCOST; vis[i][j] = 0; }
  int d = 0;
  dist[startx][starty] = 0;
  pq.push(make_pair(d, startx*Y+starty));
  while (!pq.empty()) {
    d = pq.top().first; int s = pq.top().second;
    pq.pop();
    int x = s / Y, y = s % Y;
    if (vis[x][y]) continue;
    vis[x][y] = 1;
    if (!vis[x+1][y] && !obs[x+1][y]) {
      int c = d + w(x,y,x+1,y);
      pq.push(make_pair(c,(x+1)*Y+y));
      dist[x+1][y] = min(c,dist[x+1][y]);
    }
    [...]
    if (!vis[x+1][y-1] && !obs[x+1][y-1] && (!obs[x][y-1] || !obs[x+1][y])) {
       int c = d + w(x,y,x+1,y-1);
       pq.push_back(make_pair(c,(x+1)*Y+y-1));
       dist[x+1][y-1] = min(c,dist[x+1][y-1]);
    }
    [...]
  }
}
```

2.4.1 Radix Heaps

Multi-level buckets decrease the influence on C to $\sqrt{C}$. To reduce the effect on C further, radix heaps have been proposed with an amortized worst-case complexity of $O(1)$ for *delete-min*, *decrease-key*, and $O(\lg C)$ for the *insert* operation, leading to an $O(|V|\lg C + |E|)$ shortest paths algorithm. Let C be the maximum weight of all edges. A *radix heap* maintains a list of $\lceil \lg(C+1) \rceil + 1$ buckets of sizes 1, 1, 2, 4, 8, 16, etc. Elements in the buckets are doubly-linked. The main difference to layered buckets are buckets of exponentially increasing sizes. Therefore, only $O(\lg C)$ buckets are needed. An implementation will be given in Section 16.4. There are further theoretical improvements like $\sqrt{\lg C}$ for a combination of radix with Fibonacci heaps, but for our domain, $\lg C$ is sufficiently small.

2.4.2 Bucket Maps

To bypass the dependency on C completely, a *map* of buckets can be used, for example maintained in the form of a balanced search tree. This way only buckets that are non-empty are stored and inspected for expansion. The access time, however, increases to the logarithm of the number of all buckets currently stored in the queue. In the unlikely case, each bucket contains at most one element. In principle, bucket maps work not only for integer values but also for floating-point and even real-valued data, as long as there is a total ordering.

2.4.3 Factorized Heaps

In octile grids with Euclidean distances, every path has cost $k \cdot 1 + l \cdot \sqrt{2}$ for some $k, l \geq 0$ (both k and l are bounded by the length of the optimal path L). This can be exploited in the following table of buckets.

	0	1	2	3	4	$\cdots$
0	$0+0\cdot\sqrt{2}$	$0+1\cdot\sqrt{2}$	$0+2\cdot\sqrt{2}$	$0+3\cdot\sqrt{2}$	$0+4\cdot\sqrt{2}$	$\cdots$
1	$1+0\cdot\sqrt{2}$	$1+1\cdot\sqrt{2}$	$1+2\cdot\sqrt{2}$	$1+3\cdot\sqrt{2}$	$1+4\cdot\sqrt{2}$	$\cdots$
2	$2+0\cdot\sqrt{2}$	$2+1\cdot\sqrt{2}$	$2+2\cdot\sqrt{2}$	$2+3\cdot\sqrt{2}$	$2+4\cdot\sqrt{2}$	$\cdots$
3	$3+0\cdot\sqrt{2}$	$3+1\cdot\sqrt{2}$	$3+2\cdot\sqrt{2}$	$3+3\cdot\sqrt{2}$	$3+4\cdot\sqrt{2}$	$\cdots$
4	$4+0\cdot\sqrt{2}$	$4+1\cdot\sqrt{2}$	$4+2\cdot\sqrt{2}$	$4+3\cdot\sqrt{2}$	$4+4\cdot\sqrt{2}$	$\cdots$
5	$5+0\cdot\sqrt{2}$	$5+1\cdot\sqrt{2}$	$5+2\cdot\sqrt{2}$	$5+3\cdot\sqrt{2}$	$5+4\cdot\sqrt{2}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

By processing this table we precompute an index that is used for addressing during the search. The order of buckets has to warrant increasing cost, so we sort the entries. In the worst case $L = \Omega(|V|)$, e.g., a maze in the form of a spiral. In practice, however, L is much smaller.

The order of expansion for nodes within a bucket is not important and leaves room for parallelization and externalization. As a side product, we can remove some buckets that are no longer needed, improving the memory footprint of the algorithm.

Program 2.6: Factorized shortest paths search.

```
void factorize_dijkstra(int startx, int starty) {
  int global = 0;
  for (int i=0;i<X;i++)
    for (int j=0;j<Y;j++)
      vis[i][j] = 0;
  pq[0].push_back(startx*Y+starty);
  int s = 1, n = 0;
  while (1) {
    for (int i=0;i<mpq[n].size();i++) {
      int x = mpq[n][i] / Y, y = mpq[n][i] % Y;
      s--;
      if (vis[x][y]) continue;
      vis[x][y] = 1;
      if (!vis[x+1][y] && !obs[x+1][y]) { pq[MAXLENGTH+n].push_back((x+1)*Y+y); s++; }
      [...]
      if (!vis[x+1][y-1] && !obs[x+1][y-1] && (!obs[x][y-1] || !obs[x+1][y])) { pq[1+n].push_back
          ((x+1)*Y+(y-1)); s++; }
      [...]
    }
    pq[n].clear();
    if (s==0) break;
    do {
      n = indices[++global];
    } while (pq[n].size() == 0);
  }
}
```

Precomputation reduces to sorting, and with a standard sorting algorithm for which it would take $L^2 \lg L^2 = O(|V| \lg |V|)$ time in the worst case. With multiway merging the complexity reduces to $L^2 \lg L = O(|V| \cdot \lg(\sqrt{|V|}))$, which is acceptable, as the work has to be done only once. After precomputation, Dijkstra's algorithm, as shown in Program 2.6, takes $O(|V| + K)$ time, where K is the number of buckets found empty during the search. All operations in the outer while loop are constant-time, the additional efforts in the do-while loop for pushing the pointer are $O(K)$. If $L = O(\sqrt{|V|})$ we have $K = O(L^2)$, $L^2 = O(|V|)$ and, thus, $K = O(|V|)$, so that we arrive at the optimal linear time complexity for Dijkstra's algorithm.

Program 2.7: Cache-efficient shortest paths search.

```
int MinDist(int x, int y) {
  int d = MAXCOST;
  if (dist[x-1][y]>=0 && d > dist[x-1][y]+w(x-1,y,x,y)) d = dist[x-1][y]+w(x-1,y,x,y); [...]
  if (dist[x-1][y-1]>=0 && d > dist[x-1][y-1]+w(x-1,y-1,x,y)) d = dist[x-1][y-1]+w(x-1,y-1,x,y);
      [...]
  return d;
}
int better(int x,int y) {
  return dist[x][y]>=0 && dist[x][y] > MinDist(x,y);
}
void scan(int x, int y, int step) {
  int i = y + step;
  while (better(x,i)) {
    dist[x][i] = MinDist(x,i);
    if (better(x-1,i) && !better(x-1,i-step)) queue.push((x-1)*Y+i);
    if (better(x+1,i) && !better(x+1,i-step)) queue.push((x+1)*Y+i);
    i+=step;
  }
}
void flood_fill(int startx, int starty) {
  for (int i=0;i<X;i++)
    for (int j=0;j<Y;j++)
      dist[i][j] = obs[i][j] ? -MAXCOST : MAXCOST;
  queue.push(startx*Y+starty);
  int start = startx*Y+starty;
  dist[startx][starty] = init(startx,starty);
  while (!queue.empty()) {
    int s = queue.front(); queue.pop();
    int x = s / Y, y = s % Y;
    if ((s == start) || better(x,y)) {
      if (s != start) dist[x][y] = MinDist(x,y);
      if (better(x-1,y)) queue.push((x-1)*Y+y);
      if (better(x+1,y)) queue.push((x+1)*Y+y);
    }
    scan(x,y,-1); scan(x,y,+1);
  }
}
```

2.5 Cache-Efficient Flood-Filling

In modern computers with a hierarchy of caches, localizing the search can be very effective: the retrieval time can decrease considerably if nodes are already present in a cache, rather than located in main memory.

The memory layout of nodes to be visited can result in drastic changes to the performance. In 2D image manipulation software, coloring areas is usually addressed with *flood-filling* that strictly prefers the y- to the x-direction. Speed-ups reported are in the order of one magnitude and more.

For shortest-path search, the algorithmic considerations are more challenging. The main idea of Program 2.7 is the following: when traversing the gridworld row-wise after each obstacle, the cells above or below the current cell are stored in a queue as candidates to invoke a new scan if they provide a possible improvement. This way, nodes may be reopened.

Frequently, the flood-fill shortest path algorithm computes the optimal distances for all nodes in the grid. However, there are cases, in which the algorithm in Program 2.7 does not compute the optimal solution cost value at a node.

The core problem is settling a node still to be reopened, as such settled cells (i, j) with $\text{minDist}(i, j) = dist[i][j]$ may affect the scanning. Take the following example of a small grid fragment

(x)	(x)	(e)	(f)	(x)
(x)	(a)	(b)	(c)	(x)
(x)	(x)	(x)	(d)	(g)

where we have enqueued cell (c) reaching it on a better path via cell (d) ((x) denotes a blocked cell). Before deleting cell (c) from the queue, we come from the other side via cell (e) and reach cell (b) with a worse value than the one at cell (d). While scanning to the right, cell (c) will be settled, even though it is pending in the queue. Now, when extracting cell (c) from the queue it already has the value updated, which, in turn, limits other cells (e.g., (e) and (f)) from participating in the new information.

The problem persists if cutting corners is allowed or if uniform instead of Euclidean weights are used. A fix is to prevent nodes from settling if they are contained in the queue.

This leads to the improved implementation of the flood-fill algorithm in Program 2.8 that finds optimal solutions.

Program 2.8: Flood-fill shortest path search.

```
void scan(int x, int y, int step) {
  int i = y + step; int d = MinDist(x,i);
  while (dist[x][i] >= d) {
    dist[x][i] = d;
    if (!vis[x-1][i] && better(x-1,i) && !better(x-1,i-step)) {queue.push((x-1)*Y+i); vis[x-1][i
        ]=1;}
    if (!vis[x+1][i] && better(x+1,i) && !better(x+1,i-step)) {queue.push((x+1)*Y+i); vis[x+1][i
        ]=1;}
    i += step; d = MinDist(x,i);
  }
}
void improved_flood_fill(int startx, int starty) {
  for (int i=0;i<X;i++)
    for (int j=0;j<Y;j++)
      dist[i][j] = (vis[i][j] = obs[i][j]) ? -MAXCOST : MAXCOST;
  int start = startx*Y+starty;
  queue.push(start);
  vis[startx][starty] = 1; dist[startx][starty] = 0;
  while (!queue.empty()) {
    int s = queue.front(); queue.pop(); int x = s / Y, y = s % Y;
    vis[x][y] = 0;
    int d = MinDist(x,y);
    if (s == start || dist[x][y] > d) {
      if (s != start) dist[x][y] = d;
      if (!vis[x-1][y] && better(x-1,y)) {queue.push((x-1)*Y+y); vis[x-1][y]=1;}
      if (!vis[x+1][y] && better(x+1,y)) {queue.push((x+1)*Y+y); vis[x+1][y]=1;}
    }
    scan(x,y,-1); scan(x,y,+1);
  }
}
```

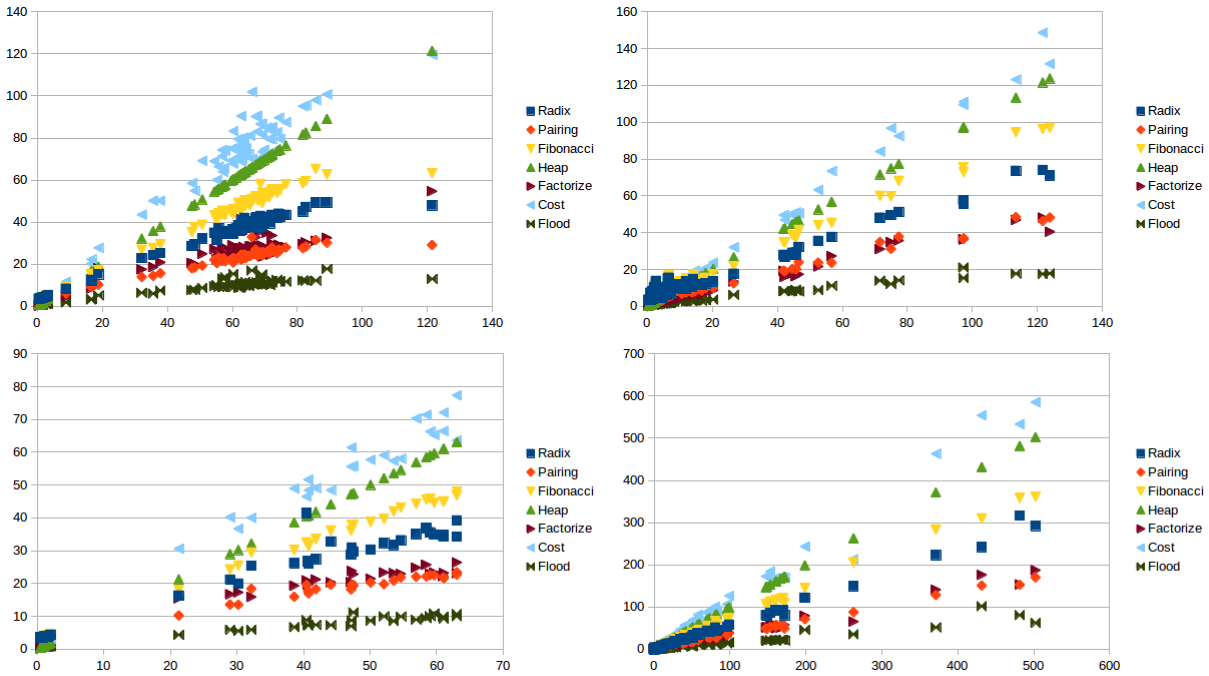


Figure 2.1: Time in milliseconds for shortest paths search in the game maps of Baldur's Gate II, Starcraft, Warcraft III, and Dragon Age (top to bottom, left to right) using scatter plots wrt the performance of shortest paths search with heaps.

2.6 Experiments

We considered two sets of problems for the experiments. All algorithms find shortest paths.

For benchmark game graphs the task lists include many s-to-t queries, so that we removed the goals t and concentrated on the efforts for a complete enumeration. (For convenience, we enforced a frame of walls).

The CPU time in Figure 2.1 is the one obtained on the first instance in which the initial cell was not blocked.

We see that improved flood-fill is generally the fastest method despite re-expanding nodes. The pairing heaps are the fastest general method and perform equally well if not better than the factorized structures. Fibonacci heaps beat binary ones, and the cost-based exploration maintaining an STL map performs worst.

2.7 Summary

In the brief study we have compared several state-of-the-art priority queues, with and without buckets, with and without domain-specific information.

In any case, shortest path search in grids is fast, and if we have access to the grid beforehand one can ask if reading the grid from disk is slower than computing the shortest paths. From a theoretical side we have that $\Omega(|V|)$ is the time for reading (and storing) the input, and some algorithms like the factorized cost scheme can achieve worst-case time $O(|V|)$ for computing all the single-source shortest path values.

Practically, there is a noticeable difference in the performance of the algorithms. As we allow slightly more complex edge costs, it is general priority queues are efficient and sometimes even outperform solutions specialized for grids. Through their good performance, cache-efficient shortest paths solutions are worth considering.

The extension of the algorithms to the 3D world of voxels are rather obvious. The outcome of the study above suggests that—depending on the setting—specialized algorithms can bu may not always result in the fastest approach. Moreover, if a mesh or a different workspace decomposition is provided as the input as an alternative discretization option, a general priority queue is the more flexible and natural implementation option.

2.8 Bibliographic Notes

Dijkstra's algorithm for solving the SSSPs problem in weighted graphs goes back to [161]. Many combinatorial optimization tasks like solving puzzles [376], or finding multiple sequence alignments [562] correspond to a search in a weighted graph.

The amortized worst case of two comparisons for *decrease-key* and $1.5 \lg n$ for *delete-min* achieved in relaxed weak-heaps [188] naturally leads to at most $2m + 1.5n \lg n + O(n)$ comparisons for finding shortest paths. Noshita has shown that on average (over perturbations of successors) at most $n \lg(m/n)$ *decrease-key* operations are executed. The two results imply linear-time average-case complexity $O(m)$ for $m = \omega(n \lg n)$ and $1.5n \lg n + O(n)$ for values $m = o(n \lg n)$. On the other hand, on sparse graphs, at least Dijkstra's SSSP method has to have complexity $\Omega(n \lg n)$, due to fixed ordering and the sorting complexity bound. In a complete graph with *simple* edge cost function SSSP high probability $\Omega(n \lg n)$ (see [473]).

If random edge weights are chosen uniformly in $[1..M]$ an average case of $O(n+m)$ (see [118]) for the SSSPs has been shown. By using a refined bucket implementation an $O(n+m)$ average-case is also valid for random edge weights chosen according to the uniform distribution on $[0,1]$ for both *label-setting* and *label-correcting algorithms* [475]. Both results come with high-probability bounds in case of independence.

The A* algorithm [323] for consistent heuristics was shown to be a variant of Dijkstra's algorithm [219]. Pattern databases [143] are effective in AI search domains such as the Rubik's Cube [414].

There are speed-up techniques for accelerating answering many *one-pair* shortest path queries including shortest-path containers, edge flags, and contraction hierarchies [638, 284]. Frequently, they call shortest path methods in the precomputation stage. Last but not least, computing all shortest paths from/to a distinguished set of nodes is a necessity in vehicle routing systems to compress large maps. This generates distance tables, serving as inputs for variants of the traveling salesman problem [292].

Regarding the data structure, there are several experimental studies on the efficiency of priority queues [118, 88, 591]. For the performance of radix heap and its possible combination with Fibonacci heaps, see [8].

The complexity time bound of $O(|E| + |V| \lg |V|)$ is supposed to be optimal for SSSP search in unrestricted graphs with a total order based on executing comparisons. Optimal time $O(|E| + |V|)$ is available on special graph classes like planar graphs [404], and edge cost functions like integers on undirected edges [613]. For SSSP search in graphs with IEEE floating point weights in directed problem edge weights, see [497].

Theoretical considerations on whether or not a traversal order in such a *localized algorithm* retains optimal solutions, have been derived [219]. Advanced general heap implementations include pairing heaps [270], Fibonacci heaps [271], Brodal heaps [85], and relaxed heaps [170]. Moreover, there are suggestions to simplify the implementation of Fibonacci heaps in [381].

In weighted graphs it has been shown that the flood-fill algorithm [450] may lead to a quadratic number of reopenings [198].

Program 2.2: Lazy simple Fibonacci heap implementation.

```
template <typename E, typename C, typename N>
class fibonacci_heap {
public:
  using element_type = E; using comparator_type = C; using node_type = N;
  using rank_type = typename N::rank_type;
  enum: rank_type {max_rank = CHAR_BIT * sizeof(N*)};
  C comparator; N* head; std::vector<N*> a;
  fibonacci_heap(C const& c = C()) : comparator(c), head(0), a(max_rank, 0) {}
  N* begin() const { return head; }
  N* end() const { return 0; }
  N* top() { if (head == 0) return 0; consolidate(); return head;}
  bool empty() { return head == 0; }
  N* inject(N* x) { head = append(x, head); return x; }
  N* eject() { N* ejected_node = extract(head); return ejected_node; }
  N* extract(N* z) {
    N* x = (*z).parent(), c = (*z).child();
    if (z == head) { N* s = (*head).after(); if (s == head) head = 0; else head = s; }
    if (x != 0) decrease_ranks(z);
    cut(z);
    if (c != 0) { N* r = c;
      do { (*r).parent(0); (*r).state(N::unmarked); r = (*r).after(); } while (r != c); }
    head = catenate(head, c); return z; }
  void increase(N* x, E const& v) {
    (*x).element() = v; N* z = (*x).parent(); if (z == 0) return;
    decrease_ranks(x); cut(x); head = append(x, head); }
protected:
  N* catenate(N* p, N* r) {
    if (p == 0) return r; if (r == 0) return p;
  N* q = (*p).before(), s = (*r).before();
    (*p).before(s); (*q).after(r); (*r).before(q); (*s).after(p); return p; }
  N* append(N* x, N* y) {
    if (y == 0) { (*x).before(x); (*x).after(x); return x; }
  N* z = (*y).before(); (*x).before(z); (*x).after(y); (*y).before(x); (*z).after(x); return y;}
  void addcirc(N* x, N* y) { N* c = (*y).child(); (*y).child(append(x, c)); (*x).parent(y);}
  void cut(N* q) {
    N* u = (*q).parent(); N* p = (*q).before(); N* r = (*q).after();
    (*q).parent(0); (*q).state(N::unmarked);
    if (p != q) { (*p).after(r); (*r).before(p); }
    if (u != 0) { if ((*u).child() == q) { if (p != q) (*u).child(r); else (*u).child(0); }}}
  void decrease_ranks(N* y) {
    do { y = (*y).parent(); if ((*y).rank() > 0) (*y).rank((*y).rank() - 1);
         (*y).state((typename N::state_type) (! (bool) (*y).state()));
    } while ((*y).state() != N::marked);
    if ((*y).parent() == 0) (*y).state(N::unmarked); }
  void consolidate() {
    rank_type border = 0; N* current = head, x;
    do {
      x = current; current = (*current).after(); N* y; rank_type d = (*x).rank();
      while (a[d] != 0) {
        y = a[d]; a[d] = 0;
        if (comparator((*x).element(), (*y).element())) std::swap(x, y);
        addcirc(y, x); (*x).rank((*x).rank() + 1); d += 1; }
      a[d] = x; if (d > border) border = d;
    } while (current != head);
    x = 0;
    for (rank_type i = 0; i <= border; ++i)
      if (a[i] != 0) {
        if (x == 0) x = a[i];
        else {
          N* y = a[i];
          if (comparator((*x).element(),(*y).element())) std::swap(x,y);
          addcirc(y, x);
        }
        a[i] = 0; }
    head = append(x, 0); }
};
```

Program 2.3: Pairing heap implementation.

```
template <typename E, typename C, typename N>
class pairing_heap {
  public:
    using element_type = E; using comparator_type = C; using node_type = N;
    C comparator; N* root;
    pairing_heap(C const& c = C()): comparator(c), root(0) {}
    N* begin() const { return root; }
    N* end() const { return 0; }
    N* top() const { return root; }
    bool empty() { return root == 0; }
    N* inject(N* f) { if (root == 0) root = f; else link(root, f); return f; }
    void increase(N* x, E const& k) {
      x->element() = k;
      N* p = x->parent(), h = 0;
      if (x != root) {
        if (x->right() != 0) { h = x->right(); h->parent(p); x->right(0); }
        if (p->left() == x) p->left(h); else p->right(h);
        x->parent(0); link(root, x); }}
    N* eject() { N* t = top(); extract(t); return t; }
    N* extract(N* x) {
      if (x == root) {
        if ((*x).left() == 0 && (*x).right() == 0) root = 0;
        else { root = root->left(); root->parent(0);
          if (root->right() != 0) root = twopass(root); }
        x->parent(0); }
      else {
        N* p = x->parent(), h = 0;
        if (x->right() != 0) { h = x->right(); h->parent(p); x->right(0); }
        if (p->left() == x) p->left(h); else p->right(h);
        x->parent(0);
        x->right(root->right());
        if (x->right() != 0) x->right()->parent(x);
        root->right(x->left());
        if (root->right() != 0) root->right()->parent(root);
        root->parent(x); x->left(root); root->parent(0);
        if (root->right() != 0) root = twopass(root);
      }
      x->parent(0); x->left(0); x->right(0);
      return x;
    }
  private:
    void link(N*& x, N*& y) {
      if (comparator(y->element(), x->element())) {
        x->right(y->right()); if (x->right() != 0) x->right()->parent(x);
        y->right(x->left()); if (y->right() != 0) y->right()->parent(y);
        y->parent(x); x->left(y);
      }
      else {
        x->right(y->left()); if (x->right() != 0) x->right()->parent(x);
        y->parent(x->parent()); if (y->parent() != 0) y->parent()->right(y);
        y->left(x); x->parent(y); x = y;
      }
    }
    N* twopass(N* h) {
      N* h1 = h, h2 = h->right();
      while (h2 != 0) {
        h = h1->right()->right(); link(h1,h2);
        if (h != 0) { if (h->right()!= 0) { h2 = h->right(); h1 = h; } else h2 = 0; }
        else { h = h1; h2 = 0; }
      }
      h1 = h->parent(); h2 = h;
      while (h1 != 0) { link(h1,h2); h = h1; h2 = h1; h1 = h1->parent(); }
      return h;
    }
};
```

Chapter 3
Sorting

Sorting a sequence of n elements probably is the most fascinating topic in computer science, and improved sorting implementations have significant impact for many applications like index calculations for databases.

We want sorting algorithms that

- are simple to implement,
- are useful in practice to be used in libraries,
- are general, so that any ordered itemset can be sorted,
- are memory-limited: *in-place* with extra space of most $O(1)$ computer words, or *in-situ* with extra space of at most $O(\lg n)$ words,
- have a low number of element comparisons ($n\lg n + O(n)$), and
- have a bounded number of other operations ($O(n\lg n)$).

3.1 Introduction

The[1] known lower bound for sequential sorting is $\lg(n!) - 1 \approx n\lg n - 1.44n + \Theta(\lg n)$ element comparisons for the worst and average case. The best upper bound for sorting is $n\lg n - 1.3n + o(n)$ element comparisons. The best-known average-case bound, however, is achieved by QuickMergesort, which will be introduced in this chapter. Using a clever partitioning scheme, and calling MergeInsertion at the end of the recursion, it requires at most $n\lg n - 1.3999n + o(n)$ element comparisons on the average ($n\lg n - 1.4n + o(n)$ for $n = 2^k$), while preserving worst-case bounds $n\lg n + O(n)$ element comparisons and $O(n\lg n)$ time for all other operations.

We will look at fast Heap-, Merge-, and Quicksort algorithms. Our interest is sorting that is

- theoretically fast: we present algorithms that are *constant-factor optimal*, i.e., that for a small constant c incur at most $n\lg n + c \cdot n + o(n)$ element comparisons.
- practically fast: we present algorithms that yield a low number of *mispredicted branches* and improve significantly over known sorting algorithms, among others including the library methods in C and Java.

For Heapsort, we start with introducing different heap structures and sifting procedures, Next, we turn to the concept of bottom heaps and to the design and efficiencies of StrongHeapsort. Then, we consider Quicksort and the even faster BlockQuicksort. Next, we look at the hybrids QuickMergesort and QuickMergeQuicksort, which are practical, but also have a small number of comparisons.

[1] This chapter is based on joint work with Armin Weiß, Amr Elmasry, and Jyrki Katajainen. It puts together and improves the work from [231, 189, 190].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_3

3.2 Heapsort

A *binary heap* is a binary tree in which each node stores one element. The elements are in *heap order*, if for each node the element stored at that node is not larger than the elements stored at its (at most) two children. A binary heap is conveniently stored as an array where the elements are kept in the breadth-first order of the tree. For a heap embedded in array $a[1,\ldots,n]$ we have $a_j \leq a_{2j}$ and $a_j \leq a_{2j+1}$.

The Heapsort algorithm has two major steps.

1. Construction: Convert an unsorted sequence of elements into a heap. For $i = \lceil n/2 \rceil$ down to 1,
   - rearrange structure rooted at node index i into a heap
2. Sorting: Sort the sequence by using the heap. While the heap is not empty
   - return minimal element in heap
   - remove minimal element from heap
   - rearrange the remaining structure into a heap

The first two steps in the sorting loop are often executed in one *extract-min* operation. For the last step as well as for iterated construction, repeated *sift-down* operations are called.

Heapsort applies at most $2n\lg n + O(n)$ element comparisons in the worst case. It is well known that the construction takes $O(n)$ time. The constant factor 2 is due to the two comparisons applied to find the minimum of three elements in each stage while sifting an element down the tree. Bottom-Up-Heapsort contributes to the fact that the expected depth is large. It executes $1.5n\lg n + O(n)$ element comparisons in the worst-case, but is better on average ($n\lg n + O(n)$ empirically and under some plausible assumptions).

The main algorithmic idea in Ultimate Heapsort leading to $n\lg n + O(n)$ comparison in the worst case is the following. In the first stage the sequence is partitioned wrt the median. Both the partitioning and the median computation step require at most $O(n)$ element comparisons (e.g., by using the median-of-median strategy). Next, Heapsort is applied to one half, where the other half is used for recursion. The gain is that this way Heapsort can borrow an element from the other half of the partitioned array being certain that it is found in the bottom-most layer. The constant for the linear term, however, is large.

3.3 Strong Heapsort

Strong heaps are binary heaps with an additional left-to-right child-dominance property and strengthened heaps are a certain mix of binary heaps at the top and strong heaps at the bottom. An in-place hybrid of a binary and several strong heaps, named *strengthened heap*, can store any multiset of elements (duplicates allowed). It operates *in-place* so that it uses $O(1)$ extra space in addition to the elements maintained at the beginning of an array. It supports *minimum* in $O(1)$ worst-case time with no element comparisons; *construct* in $O(n)$ worst-case time involving at most $(23/12)n + o(n)$ element comparisons; and *extract-min* in $O(\lg n)$ worst-case time involving at most $\lceil \lg n \rceil + 1$ element comparisons.

We will see that when a strengthened heap is used in Heapsort, the resulting algorithm sorts n elements in-place in $O(n\lg n)$ worst-case time involving at most $n\lg n + cn + o(n)$ element comparisons with $c \approx 1$.

In StrongHeapsort the main heap has two layers: a *top heap* that is a binary heap, and each leaf of the top heap roots a *bottom heap* that is a complete strong heap. The main heap is laid out in the array as a binary heap and, in accordance, every bottom heap is scattered throughout the array.

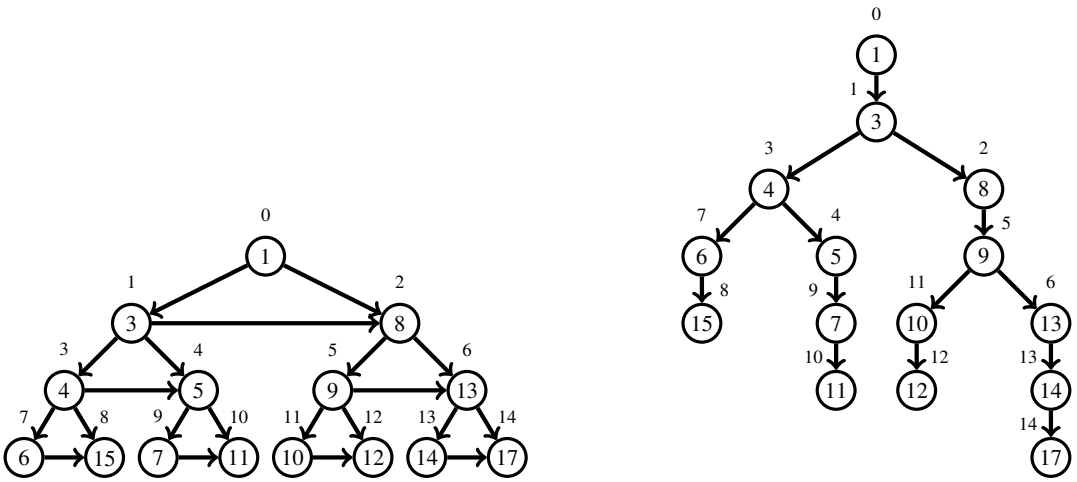


Figure 3.1: A strong heap in an array $\mathbf{a}[0:14] = [1,3,8,4,5,9,13,6,15,7,11,10,12,14,17]$ viewed as a directed acyclic graph (left) and a stretched tree (right)

The main new ingredient is the *border* maintained between the top heap and the bottom heaps. When the data structure contains n elements, the *height* h_0 of the top heap is set to $\lceil \lg n - \lg\lg n \rceil$. We represent it using one interval (ℓ, r) which is completely determined by h_0 and vice versa. We dropped the latter, and computed it on demand. The variables needed by a strengthened heap are n for the index **a** for the elements, *border* as a *struct* of indices ℓ and r.

Strong Heapsort is fully in-place. To improve the comparison bound for *extract-min*, we reinforce a stronger heap order at the bottom levels of the heap.

3.3.1 Strong Heap Construction

A *strong heap* is a binary heap with one additional invariant: The element at any right child is not smaller than that at the left sibling. This left-dominance property is fulfilled for every right child in a *fine heap*, which uses one extra bit per node to maintain the property. On the contrary, a strong heap operates in-place, but its operations are slower. Like a binary heap, a strong heap is viewed as a nearly-complete binary tree where the lowest level may be missing some nodes at the rightmost positions. In addition, this tree is embedded in an array in the same way, i.e., the formulas for computing *left-child*, *right-child*, and *parent* are still the same.

Two views of a strong heap are exemplified in Figure 3.1. On the left, the directed acyclic graph has a nearly-complete binary tree as its *skeleton*: There are arcs from every parent to its children and additional arcs from every left child to its sibling indicating the dominance relations. On the right, in the *stretched tree*, the arcs from each parent to its right child are removed as these dominance relations can be induced. In the stretched tree a node can have 0, 1, or 2 children. A node has one child if in the skeleton it is a right child that is not a leaf or a leaf that has a right sibling. A node has two children if in the skeleton it is a left child that is not a leaf. If the skeleton has height h (height of a single node being 1), the height of the stretched tree is at most $2h-1$, and on any root-to-leaf path in the stretched tree the number of nodes with two children is at most $h-2$.

When implementing *construct* and *extract-min* for a binary heap, the basic primitive used is *sift-down*. For a strong heap, *strengthening-sift-down* has the same purpose, and the implementation (see Algorithm 3.1) is similar, with one crucial exception that we operate with the stretched tree instead of the nearly-complete binary tree. As for a binary heap, *extract-min* can be implemented (also in Algorithm 3.1) by replacing the element at the root with the element at the last position of the array (if there is any) and then invoking *strengthening-sift-down*

Algorithm 3.1: Implementation of *strengthening-sift-down*, and its use in *construct* and *extract-min*.

```
procedure sibling
input i: index
output index
if i = 0 :
  return 0
return i + even(i+1) − even(i)

procedure is-leaf
input i: index, n: index
output Boolean
if not even(i) :
  return sibling(i) ≥ n
return left-child(i) ≥ n

procedure strengthening-sift-down
input i: index, n: index
x ← a[i]
while not is-leaf(i,n) :
  j ← sibling(i)
  if even(i) :
    j ← left-child(i)
  else if j < n and left-child(i) < n and not (a[j] < a[left-child(i)]) :
    j ← left-child(i)
  if not (a[j] < x) :
    break
  a[i] ← a[j]
  i ← j
a[i] ← x
```

```
procedure construct
input m: index, b: element[ ] as reference
n ← m
a ← b
if n < 2 :
  return
for i ← n − 1 down to 0 :
  strengthening-sift-down(i,n)

procedure extract-min
n ← n − 1
if n ≠ 0 :
  a[0] ← a[n]
  strengthening-sift-down(0,n)
```

for the root. Consider the strong heap in Figure 3.1. If its minimum was replaced with the element 17 taken from the end of the array, the path to be followed by *strengthening-sift-down* would include the nodes labeled 3, 4, 5, 7, and 11.

Let n denote the size of a strong heap and h the height of the underlying tree skeleton. When going down the stretched tree, we have to perform at most $h-2$ element comparisons due to branching at binary nodes and at most $2h-1$ element comparisons due to checking whether to stop or not. Hence, the number of element comparisons performed by *extract-min* is bounded by $3h-3$, which is at most $3\lg n$ as $h = \lfloor \lg n \rfloor + 1$.

To build a strong heap, we mimic Floyd's heap-construction algorithm; that is, we invoke *strengthening-sift-down* for all nodes, one by one, processing them in reverse order of their array positions (see Algorithm 3.1). One element comparison is needed for every met left child in order to compare the element at its right sibling with that at its left child, making a total of at most $n/2$ element comparisons. The number of other element comparisons is bounded by the sum $\sum_{i=1}^{\lfloor \lg n \rfloor + 1} 3 \cdot i \cdot \lceil n/2^{i+1} \rceil$, which is at most $3n + o(n)$. Thus, *construct* requires at most $3.5n + o(n)$ element comparisons.

For *construct* and *extract-min*, the amount of work done is proportional to the number of element comparisons performed, i.e., the worst-case running time of *construct* is $O(n)$ and that of *extract-min* is $O(\lg n)$. This means that a strong heap of size n can be built in $O(n)$ worst-case time by repeatedly calling *strengthening-sift-down*. Each *strengthening-sift-down* uses $O(\lg n)$ worst-case time and performs at most $3\lg n$ element comparisons.

Next we show how to perform a *sift-down* operation on a strong heap of size n with at most $\lg n + O(1)$ element comparisons. At this stage we allow the amount of work to be higher, namely $O(n)$. To achieve the better comparison bound, we have to assume that the heap is *complete*, i.e., that all leaves have the same depth. Consider the case where the element at the root of a strong heap is replaced by a new element. To reestablish strong heap order, the *swapping-sift-down* procedure (Algorithm 3.2) traverses the left spine of the skeleton bottom up starting from the leftmost leaf, and determines the correct place of the new element, using one element

Algorithm 3.2: Implementation of *swapping-sift-down* for a complete strong heap.

```
procedure swap-subtrees
input u: index, v: index, n: index
j ← 1
while v < n :
  for i ← 0 to j − 1 :
    swap(a[u+i], a[v+i])
  u ← left-child(u)
  v ← left-child(v)
  j ← 2 ∗ j

procedure leftmost-leaf
input i: index, n: index
output index
while left-child(i) < n :
  i ← left-child(i)
return i
```

```
procedure bottom-up-search
input i: index, j: index
output index
while i < j and not (a[i] < a[j]) :
  j ← parent(j)
return j

procedure swapping-sift-down
input i: index, n: index
k ← leftmost-leaf(i,n)
k ← bottom-up-search(i,k)
lift-up(i,k,n)
```

comparison at each node visited. Thereafter, it moves all the elements above this position on the left spine one level up and inserts the new element into its correct place. If this place is at height g, we have performed g element comparisons. Up along the left spine there are $\lg n - g + O(1)$ remaining levels to which we have moved other elements. While this results in a heap, we still have to reinforce the left-dominance property at these upper levels. In accordance, we compare each element that has moved up with the element at the right sibling. If the element at index j is larger than the element at index $j+1$, we interchange the subtrees T_j and T_{j+1} rooted at positions j and $j+1$ by swapping all their elements. The procedure continues until the root is reached.

Consider the strong heap in Figure 3.1. If the element at the root was replaced with the element 16, the left spine to be followed by *swapping-sift-down* would include the nodes labeled with 3, 4, and 6, the new element would be placed at the last leaf we ended up with, the elements on the left spine would be lifted up one level, and an interchange would be necessary for the subtrees rooted at node 6 and its new sibling 5.

Given two complete subtrees of height h, the number of element moves needed to interchange the subtrees is $O(2^h)$. As $\sum_{h=1}^{\lfloor \lg n \rfloor} O(2^h)$ is $O(n)$, the total work done in the subtree interchanges is $O(n)$. Thus, in a complete strong heap of size n, *swapping-sift-down* runs in-place and uses at most $\lg n + O(1)$ element comparisons and $O(n)$ moves.

To improve *construct* with respect to the number of element comparisons and the number of element moves performed, still keeping the worst-case running time linear, we can use the algorithms developed for a fine heap. Instead of *swapping-sift-down*, the subtree interchanges are realized by flipping the dominance bits. The basic algorithm builds a fine heap of size n in $O(n)$ worst case time with at most $2n + o(n)$ element comparisons. An interesting change in the base case leads to an improvement reducing the number of element comparisons performed to $(23/12)n + o(n)$. In accordance, for a strong heap, we would expect an in-place construction algorithm with at most $(23/12)n + o(n)$ element comparisons. For a strong heap we want the elements to be placed at their correct final locations, while for a fine heap it is the case that guiding information encoded in bits is sufficient. To this extent, we have to transform a fine heap into a strong heap.

In an array representation of a heap (assume the position of the root is 1), the position of a node x in binary $\psi(x) = (b_0\ b_1\ b_2\ \ldots)_2$ determines the path to that particular node from the root. We have b_0 always 1, and b_i determines the branching at level $i-1$ in the tree to reach x. Let r be the array of rotation bits of the fine heap, and let $s(x)$ be the sequence of rotation bits along the path from the root to the node x. For a linear-time algorithm, we make the following observation. The sequence $s(x)$ is uniquely determined by the array r and $\psi(x)$. Moreover, the position $\phi(x)$ for x to go in the rotated tree is determined by the bitwise xor ($\otimes$) of $\psi(x)$ and $s(x)$.

Algorithm 3.3: Transformation of a complete fine heap into a strong heap.

```
procedure scan
for i ← 1 to n :
  visited[i − 1] ← false
for i ← n downto 1 :
  while ¬visited[i − 1] :
    y ← i
    t ← a[i − 1]
    do:
      visited[i − 1] ← true
      swap(t, a[φ(i) − 1])
      i ← φ(i)
    while i ≠ y
    a[i − 1] ← t
```

Suppose we are given array $a[0..6] = (a_0|a_1,a_2|a_3,a_4,a_5,a_6)$, where $|$ separates the levels of the tree. Suppose also we have the sequence of rotation bits $r[0..6] = (1010000)$. This combination of a and r should be derotated into a strong heap $a'[0..6] = (a_0|a_2,a_1|a_6,a_5,a_3,a_4)$. For level 2 we have

- $\psi(a_3) = 4 = (100)_2$ and $s(a_3) = (10)_2$, $(1(0 \otimes 1)(0 \otimes 0))_2 = (110)_2 = 6$, so that $\phi(4) = 6$.
- $\psi(a_4) = 5 = (101)_2$ and $s(a_4) = (10)_2$, $(1(0 \otimes 1)(1 \otimes 0))_2 = (111)_2 = 7$, so that $\phi(5) = 7$.
- $\psi(a_5) = 6 = (110)_2$ and $s(a_5) = (11)_2$, $(1(1 \otimes 1)(0 \otimes 1))_2 = (101)_2 = 5$, so that $\phi(6) = 5$.
- $\psi(a_6) = 7 = (111)_2$ and $s(a_6) = (11)_2$, $(1(1 \otimes 1)(1 \otimes 1))_2 = (100)_2 = 4$, so that $\phi(7) = 4$.

Knowing that $\phi(x)$ is uniquely determined by $\psi(x)$ and r, and that $\psi(x)$ and $\phi(x)$ use $O(\lg\lg\lg n)$ bits while r has $(1/2)\lg\lg n$ bits, we can build a look-up table for all possible values of $\psi(x)$ and r together with the corresponding $\phi(x)$, all in less than $\lg n$ bits (one computer word).

In Algorithm 3.3, we scan the fine heap layer-by-layer from left to right and move elements within their permutation cycle, marking all nodes visited by setting a bit. We continue with the element at the destination of the previous move. If a cycle of swaps is completed, we progress the scan. If an element has been moved already, we continue scanning with the next element.

The results presented in this section can be summarized as follows. Let n denote the number of elements stored in a data structure. A strong heap is an in-place priority queue data structure, for which *construct* requires $O(n)$ worst-case time performing at most $(23/12)n + o(n)$ element comparisons; *minimum* requires $O(1)$ worst-case time involving no element comparisons; and *extract-min* requires $O(\lg n)$ worst-case time involving at most $3\lg n$ element comparisons; it can be refined to perform $\lg n + O(1)$ element comparisons, but this would increase its worst-case running time to $O(n)$.

3.3.2 Sorting with Strengthened Heaps

Now the main heap has two layers: a *top heap* that is a binary heap, and each leaf of the top heap roots a *bottom heap* that is a complete strong heap. The main heap is laid out in the array as a binary heap and, in accordance, every bottom heap is scattered throughout the array.

The main new ingredient is the *border* maintained between the top heap and the bottom heaps. When the data structure contains n elements, the *height* h_0 of the top heap is set to $\lceil \lg n - \lg\lg n \rceil$. We represent it using one interval (ℓ, r) which is completely determined by h_0 and vice versa. We dropped the latter, and compute it on

Algorithm 3.4: Description of the iterative processes to lift-up the border.

```
procedure raise-border
k ← sibling(border.r)
j ← border.ℓ
border.r ← parent(border.r)
border.l ← parent(border.l)
for i ← k down to j step −2 :
  strengthening-sift-down(i,n)
```

Algorithm 3.5: Implementation of Strong Heapsort.

```
procedure ancestor
input i: index, d: height
output index
return ⌊(i+1)/2^d⌋ − 1

procedure rotate
input i: index, k: index, h: height
x ← a[i]
for d ← h−1, h−2, ..., 0 :
  a[ancestor(k,d+1)] ← a[ancestor(k,d)]
a[k] ← x

procedure correct-place
input i: index, k: index, h: height
output index
d ← h
while i ≠ k :
  h′ ← ⌊(h+1)/2⌋
  j ← ancestor(k,h′)
  h ← h−h′
  if a[i] ≤ a[j] :
    k ← j
    d ← d−h′
  else:
    i ← ancestor(k,h)
return (i,d)
```

```
procedure binary-search-sift-up
input i: index, k: index, h: height
(j,d) ← correct-place(i,k,h)
rotate(i,j,d)

procedure is-on-border
input i: index
output Boolean
return border.ℓ ≤ i and i ≤ border.r

procedure combined-sift-down
input i: index, n: index
j ← i
while not is-on-border(j) :
  k ← left-child(j)
  if a[sibling(k)] < a[k] :
    k ← sibling(k)
  j ← k
if a[i] ≤ a[j] :
  binary-search-sift-up(i,parent(j),h−1)
else:
  rotate(i,j,h)
  swapping-sift-down(j,n)
```

demand. The variables needed by a strengthened heap are n for the index **a** for the elements, *border* as a *struct* of indices ℓ and r.

To raise the border, we scan the nodes on the old one and apply the operation *strengthening-sift-down* on each left child (see the procedure *raise-border* in Algorithm 3.4). Again, we only need a constant amount of space to record the state of this process. In connection with every modifying operation, if $n = 2^h - 1$ for some non-negative integer h, we check the relationship between h and h_0. If they are equal, we know that there is no need to move the border. The total work involved in border raising is linear.

To improve the performance of *extract-min* in the main heap, we use a new procedure, which we call *combined-sift-down* (Algorithm 3.5), instead of *sift-down*. Assume we have to replace the minimum of the main heap with another element. To reestablish heap order, we apply stopover optimization: We traverse down along the special path until we reach a root of a bottom heap. By comparing the replacement element with the element at that root, we check whether the replacement element should land in the top heap or in the bottom heap. In the first case, in *binary-search-sift-up*, we find the position of the replacement element using binary search on the traversed path and thereafter do the required element moves. In the second case, we apply *swapping-sift-down* on the root of the bottom heap.

Unfortunately, *swapping-sift-down* only works for complete bottom trees. Of the entire set of bottom trees there is only one with root r of size $\lceil \lg n \rceil$ that is critical. For a given value of n the root r can be determined in $O(1)$ time. For the nodes in this tree we invest a word for (fine-heap) bits that denotes for each internal node, if the subtree below it is rotated or not. This way we do not have to swap subtrees. If the root moves due to several *extract-min* operations, we derotate the entire subtree rooted at r, so that it becomes a strong heap again.

Derotation takes no comparisons but the number of element moves rises rapidly. Naively implementing derotation yields a running time quadratic in the size of the bottom tree. As with the construction of a fine-heap, it is possible to reduce the running time to linear in the size of the tree. The running time amortizes in a sequence of *extract-min* operations.

Let n denote the number of elements stored in a data structure. A strengthened heap is an in-place priority queue, for which

(1) *construct* requires $O(n)$ worst-case time performing at most $(23/12)n + o(n)$ element comparisons;

(2) *minimum* requires $O(1)$ worst-case time; and no comparison

(3) *extract-min* requires amortized $O(\lg n)$ worst-case time involving at most $\lg n + 1$ element comparisons.

Next, we look at sorting with strong heaps. A heap on i elements has depth $\lfloor \lg i \rfloor$. For the call to *combined-sift-down* we distinguish two cases.

binary-search-sift-up We know that binary search executed on n elements induces at most $1 + \lfloor \lg n \rfloor$ element comparisons. The total number of comparisons in *combined-sift-down* is the sum of the search efforts to the border plus the efforts for the binary bottom-up search. Together ($n \geq 5$) we have less than $\lfloor \lg n \rfloor - \lceil \lg \lg n \rceil + \lfloor \lg(\lfloor \lg n \rfloor - \lceil \lg \lg n \rceil) \rfloor + 1 \leq \lfloor \lg n \rfloor - \lg \lg n + \lg(\lg n - \lg \lg n) + 1 \leq \lfloor \lg n \rfloor + 1 = depth(n) + 1$ element comparisons.

swapping-sift-down The height of the remaining tree is $h = \lfloor \lg \lg n \rfloor$. The number of comparisons is bounded by h (we have to compare with the siblings at each height and invoke a swapping sift-down) This results in a number of comparisons, that is at most $1 + \lfloor \lg n \rfloor - \lceil \lg \lg n \rceil + \lceil \lg \lg n \rceil \leq \lfloor \lg n \rfloor + 1 = depth(n) + 1$.

Hence, applying *combined-sift-down* to a heap of size i takes at most $depth(i) + 1$ comparisons.

The sum of the depth of all heaps in the selection stage is $A(n) = \sum_{i=1}^{\lfloor \lg n \rfloor - 1} i2^i + \lfloor \lg n \rfloor (n - 2^{\lfloor \lg n \rfloor} + 1) = n\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + \lfloor \lg n \rfloor + 2$. In the interval(s) $[2^k, 2^{k+1})$ function $f(n) = \lg n - \lfloor \lg n \rfloor + 2^{\lfloor \lg n \rfloor + 1}/n$ takes its minimum at $n_0 = (2 \ln 2)2^k$ with $f(n_0) = 1.913828$. Thus, if $A(n)$ is the sum of depths of all heaps in the selection stage, we have $A(n) \leq n \lg n - 1.913n + o(n)$.

The number of element comparisons is the sum of construction, selection, or border lifting.

- Heap construction can be done in-place by using the method described above. The efforts for construction are, thus, bounded by $23/12n + o(n)$ element comparisons.
- The efforts for the selection stage are bounded by calling the *extract-min* operation n times, which boils down to call *combined-sift-down* to a heap of decreasing size. Let $A(n)$ of the sum of depths of all heaps in the selection phase. We have seen that $A(n)$ is bounded from above by $n \lg n - 1.913n + o(n)$. By the argument above the number of comparisons of all *combined-sift-down* operations in the selection phase is $n \lg n - 0.913n + o(n)$.
- From the analysis of heap construction in standard Heapsort for a heap with $n = 2^h$ we know that $\sum_{i=0}^{h} i \cdot 2^{h-i} = 2^h(\sum_{i=0}^{h} i/2^i) \leq 2n$. Similarly, the efforts of all calls to border-lifting are linear in $2^h(\sum_{i=g}^{h} i/2^i)$ for $g = \lg \lg n + O(1)$ and $h = \lg n + O(1)$. Since $\sum_{i=k}^{\infty} i/2^i = 2(k+1)/2^k$ we have $\sum_{i=\lg \lg n}^{\lg n} i/2^i \leq 2(\lg \lg n + 1)/2^{\lg \lg n} = 2(\lg \lg n + 1)/\lg n \to 0$ for $n \to \infty$. Adding $O(1)$ to either term does not affect this truth. As $\sum_{i=g}^{h} i/2^i = o(1)$, for $g = \lg \lg n + O(1)$ and $h = \lg n + O(1)$ we have $2^h(\sum_{i=g}^{h} i/2^i) = o(n)$.

Let n denote the number of elements to be sorted. Putting things together for sorting based on strenthened heaps we have at most $23/12n + o(n) + n\lg n - 0.913n + o(n) + o(n) = n\lg n + 1.01n + o(n)$ element comparisons. Both construction and selection are in-place.

3.4 Improving Quicksort

Quicksort is commonly considered as one of the fastest (in-situ) sorting algorithms. The central part of Quicksort is the partitioning procedure. Given some pivot element, it returns a pointer p to an element in the array and rearranges it, such that all elements left of the p are smaller than or equal to the pivot and all elements on the right are greater than or equal to the pivot.

Quicksort first chooses some pivot element, then performs the partitioning, and, finally, recurses on the elements smaller and the elements larger than the pivot. We call the procedure which organizes the calls to the partitioner the *Quicksort main loop*.

A very basic optimization avoids recursion partially or totally (this requires the introduction of an explicit stack). Introsort uses an additional counter for the number of recursion levels. As soon as it exceeds some bound, the algorithm stops Quicksort and switches to Heapsort (only for the respective sub-array). By doing so, a worst-case running time of $O(n\lg n)$ is guaranteed.

One may also swap to Insertionsort as soon as the array size is less than some fixed small constant. There are two possibilities when to apply Insertionsort: either during the recursion, when the array size becomes too small, or at the very end after Quicksort has finished.

The basic version of Quicksort uses a random or fixed element as pivot. A slight improvement is to choose the pivot as the median of three elements – typically the first, in the middle and the last. This is applied in Introsort and many other Quicksort implementations.

Choosing the pivots from a larger sample does not provide a significant increase of speed, even if the number of comparisons in Quicksort is minimal if the pivot element is selected as median of $\Theta(\sqrt{n})$ elements. After the partitioning, the pivot is moved to its correct position and not included into the recursive calls.

Another development is *multi-pivot Quicksort* (i.e., several pivots in each partitioning stage). It started with the introduction of Yaroslavskiy's dual-pivot Quicksort – which, surprisingly, was faster than known Quicksort variants and, thus, became the standard sorting implementation in Oracle Java 7 and 8.

Concerning branch mispredictions, multi-pivot Quicksort variants all behave essentially the same way as ordinary Quicksort; however, they have one advantage: every data element is accessed only fewer times (this is also referred to as the number of *scans*).

Increasing the number of pivot elements further (up to 127 or 255), leads to SuperScalarSampleSort. SuperScalarSampleSort not only has the advantage of few scans, but also is based on the idea of avoiding conditional branches. Indeed, the correct bucket (the position between two pivot elements) can be found only by converting the results of comparisons to integers and then performing integer arithmetic. SuperScalarSampleSort is approximately twice as fast as Introsort but has one major draw-back: it uses a linear amount of extra space (for sorting n data elements, it requires space for another n data elements and additionally for more than n integers).

3.5 Block Quicksort

The idea of block partitioning is quite simple. Recall Hoare's original partition procedure (Algorithm 3.6): Two pointers start at the leftmost and rightmost elements of the array and move towards the middle. In every

Algorithm 3.6: Hoare's partitioning.

```
procedure Partition(A[ℓ,...,r], pivot)
while ℓ < r :
  while A[ℓ] < pivot :
    ℓ++
  while A[r] > pivot :
    r--
  if ℓ < r :
    swap(A[ℓ],A[r])
    ℓ++
    r--
return ℓ
```

step the current element is compared to the pivot. If some element on the right side is less than or equal to the pivot or some element on the left side is greater than or equal to the pivot, the respective pointer stops and the two elements found this way are swapped. Then the pointers continue moving towards the middle.

The idea of BlockQuicksort (Algorithm 3.7) is to split Algorithm 3.6 in two stages: fix some block size B; we introduce two buffers $\text{offsets}_L[0,\ldots,B-1]$ and $\text{offsets}_R[0,\ldots,B-1]$ for storing pointers to elements (offsets_L will store pointers to elements on the left side of the array which are greater than or equal to the pivot element – likewise offsets_R for the right side). The main loop of Algorithm 3.7 consists of two phases: the scan phase and the rearrange phase.

As for classical Hoare partition, we also start with two pointers (or indices as in the pseudo-code) to the leftmost and rightmost element of the array. First, the scan phase takes place: the buffers which are empty are refilled. In order to do so, we move the respective pointer towards the middle and compare each element with the pivot. However, instead of stopping at the first element which should be swapped, only a pointer to the element is stored in the respective buffer (the pointer is always stored, but depending on the outcome of the comparison, a counter holding the number of pointers in the buffer is increased or not) and the pointer continues moving towards the middle. After an entire block of B elements has been scanned (either on both sides of the array or only on one side), the rearranging phase begins: it starts with the first positions of the two buffers and swaps the data elements they point to; then it continues until one of the buffers contains no more pointers to elements which should be swapped. Now the scan phase is restarted and the buffer that has run empty is filled again.

The algorithm continues this way until there remain fewer elements than two times the block size. Now, the simplest variant is to switch to the usual Hoare partition method for the remaining elements. But we also can continue with the idea of block partitioning: the algorithm scans the remaining elements as one or two final blocks (of smaller size) and performs a last rearrange phase. After that, there might still remain some elements to swap in one of the two buffers, while the other buffer is empty. With one run through the buffer, all these elements can be moved to the left or right (as it is done in the Lomuto partitioning method, but without performing actual comparisons). We do not present the details for this final rearranging here because on one hand it gets a little tedious and on the other hand it neither provides a lot of insight into the algorithm nor is it necessary to prove the result on branch mispredictions. For an example with an array of 12 elements, see Figure 3.2. One possible C++ implementation is given in Programs 3.1–3.2.

If the input consists of random permutations, the average numbers of comparisons and swaps are the same as for usual Quicksort with median-of-three. This is because both Hoare's partitioner and the block partitioner preserve randomness of the array.

The number of scanned elements (total number of elements loaded to the registers) is increased by two times the number of swaps, because for every swap, the data elements have to be loaded again. However, the idea is that due to the small block size, the data elements still remain in L1 cache when being swapped – so the additional scan has no negative effect on the running time. For larger data types and from a certain threshold on, an increasing size of the blocks has a negative effect on the running time. Therefore, the block size should

Algorithm 3.7: Block partitioning.

```
procedure BlockPartition A[ℓ,...,r], pivot
offsets_L[0,...,B−1], offsets_R[0,...,B−1]
start_L, start_R ← num_L ← num_R ← 0
while r − ℓ + 1 > 2B :
  if num_L ≠ 0 :
    start_L ← 0
    for i = 0,...,B−1 :
      offsets_L[num_L] ← i
      num_L +=(pivot ≥ A[ℓ+i])
  if num_R ≠ 0 :
    start_R ← 0
    for i = 0,...,B−1 :
      offsets_R[num_R] ← i
      num_R +=(pivot ≤ A[r−i])
  num ← min(num_L, num_R)
  for j = 0,...,num−1 :
    swap(A[ℓ+offsets_L[j]], A[r−offsets_R[j]])
  num_L, num_R −= num
  start_L, start_R += num
  if (num_L = 0) :
    ℓ += B
  if (num_R = 0) :
    r −= B
compare and rearrange remaining elements
```

not be chosen too large – we suggest $B = 128$ (thus, already for inputs of moderate size, the buffers also do not require much more space than the stack for Quicksort).

In Figure 3.3 we illustrate the outcome of cross-comparing BlockQuicksort with other known efficient sorting algorithms, including the library implementations in Java and C++.

3.6 Quick Mergesort

Now we turn to another sorting approach that is aimed at reducing the number of comparisons. All algorithms we present will be hybrids of established sorting schemes.

QuickXSort

Let X be some sorting algorithm. QuickXsort works as follows: First, choose a pivot element as the median of some random sample. Next, partition the array according to this pivot element, i. e., rearrange the array such that all elements left of the pivot are less than or equal to the pivot element, and all elements on the right are greater than or equal to the pivot element. (If the algorithm X outputs the sorted sequence in the extra memory, the partitioning is performed such that the all elements left of the pivot are greater or equal and all elements on the right are less or equal than the pivot element.) Then, choose one part of the array and sort it with algorithm X. (The preferred choice depends on the sorting algorithm X.) After one part of the array has been sorted with X, move the pivot element to its correct position (right after/before the already sorted part) and sort the other part of the array recursively.

The main advantage of this procedure is that the part of the array that is not being sorted currently can be used as temporary memory for the algorithm X. This yields fast *internal* variants for various *external* sorting algorithms

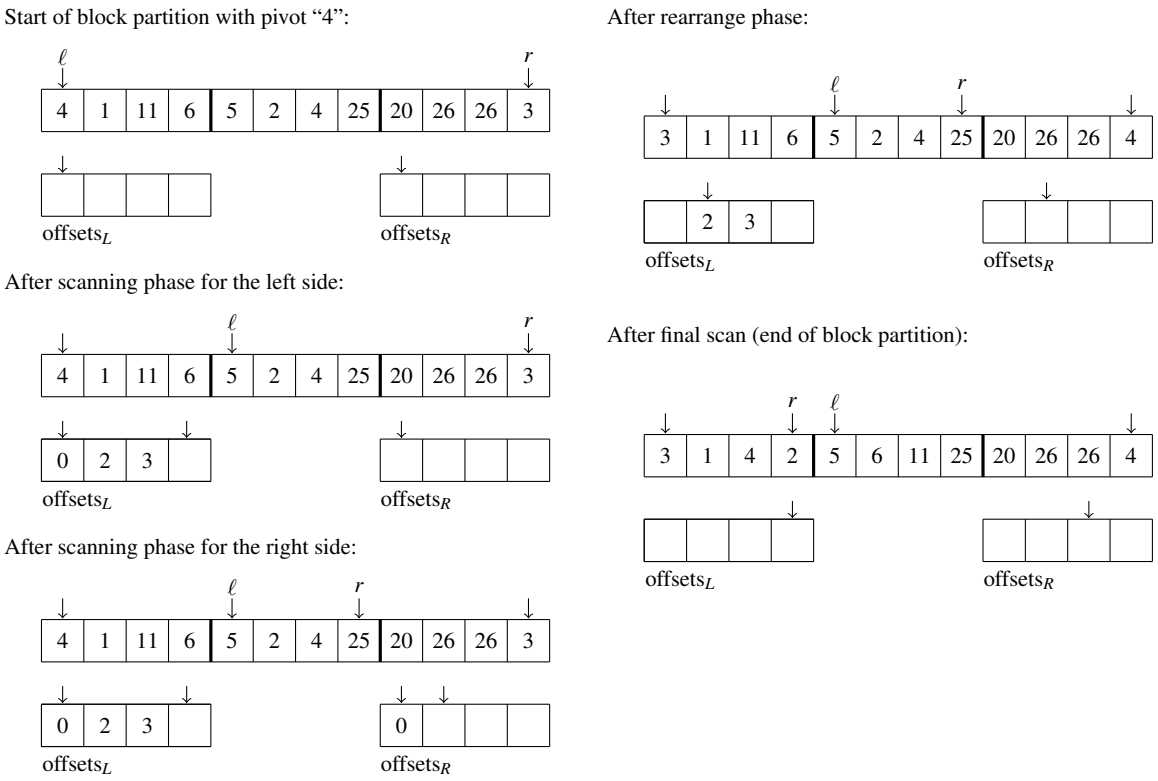


Figure 3.2: Example of block partitioning with block size $B = 4$ and pivot “D”.

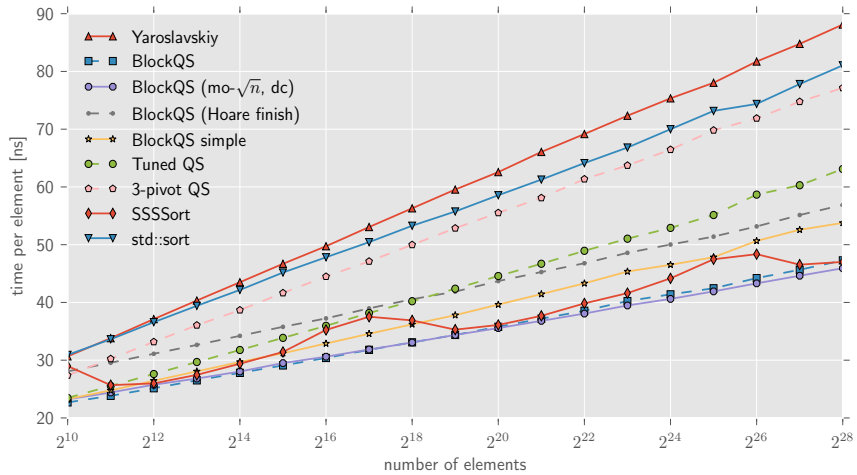


Figure 3.3: CPU time for block-sorting random integer data of increasing size.

such as Mergesort. The idea is that whenever a data element should be moved to the external storage, instead it is swapped with the data element occupying the respective position in part of the array which is used as temporary memory.

Program 3.1: Implementation of partitioner in BlockQuicksort.

```
template<typename iter, typename Compare>
inline iter hoare_block_partition_simple(iter begin, iter end, iter pivot_position,
    Compare less) {
  typedef typename iterator_traits<iter>::difference_type index;
  index i_l[BLOCKSIZE], i_r[BLOCKSIZE]; iter last = end - 1;
  iter_swap(pivot_position, last);
  const typename iterator_traits<iter>::value_type & pivot = *last;
  pivot_position = last--;
  int n_l = 0, n_r = 0, start_l = 0, start_r = 0, num;
  while (last - begin + 1 > 2 * BLOCKSIZE) {  //main loop
    if (n_l == 0) {  //Compare and store in buffers
    start_l = 0;
    for (index j = 0; j < BLOCKSIZE; j++) { i_l[n_l] = j; n_l += (!(less(begin[j], pivot))); }}
    if (n_r == 0) {
    start_r = 0;
    for (index j = 0; j < BLOCKSIZE; j++) { i_r[n_r] = j; n_r += !(less(pivot, *(last - j))); }}
    num = min(n_l, n_r);  //rearrange elements
    for (int j = 0; j < num; j++) iter_swap(begin + i_l[start_l + j], last - i_r[start_r + j]);
    n_l -= num; n_r -= num; start_l += num; start_r += num;
    begin += (n_l == 0) ? BLOCKSIZE : 0; last -= (n_r == 0) ? BLOCKSIZE : 0; }
  index s_r = 0, s_l = 0; // compare and store in buffers final iteration
  if (n_r == 0 && n_l == 0) { // for small arrays or unlikely case that both buffers are empty
    s_l = ((last - begin) + 1) / 2; s_r = (last - begin) + 1 - s_l; start_l = 0; start_r = 0;
    for (index j = 0; j < s_l; j++) {
      i_l[n_l] = j; n_l += (!less(begin[j], pivot)); i_r[n_r] = j; n_r += !less(pivot, *(last-
          j)); }
    if (s_l < s_r) { i_r[n_r] = s_r - 1; n_r += !less(pivot, *(last - s_r + 1)); }}
  else if (n_r != 0) {
    s_l = (last - begin) - BLOCKSIZE + 1; s_r = BLOCKSIZE; start_l = 0;
    for (index j = 0; j < s_l; j++) { i_l[n_l] = j; n_l += (!less(begin[j], pivot)); }}
  else {
    s_l = BLOCKSIZE; s_r = (last - begin) - BLOCKSIZE + 1; start_r = 0;
    for (index j = 0; j < s_r; j++) { i_r[n_r] = j; n_r += !(less(pivot, *(last - j))); }}
  num = min(n_l, n_r); //rearrange final iteration
  for (int j = 0; j < num; j++)
    iter_swap(begin + i_l[start_l + j], last - i_r[start_r + j]);
  n_l -= num; n_r -= num; start_l += num; start_r += num;
  begin += (n_l == 0) ? s_l : 0; last -= (n_r == 0) ? s_r : 0;
  if (n_l != 0) { // end final iteration, rearrange elements remaining in buffer
    int low_I = start_l + n_l - 1; index upper = last - begin;
    while (low_I >= start_l && i_l[low_I] == upper) { upper--; low_I--; }
    while (low_I >= start_l)
      iter_swap(begin + upper--, begin + i_l[low_I--]);
    iter_swap(pivot_position, begin + upper + 1); // fetch the pivot
    return begin + upper + 1; }
  else if (n_r != 0) {
    int low_I = start_r + n_r - 1; index upper = last - begin;
    while (low_I >= start_r && i_r[low_I] == upper) { upper--; low_I--; }
    while (low_I >= start_r) iter_swap(last - upper--, last - i_r[low_I--]);
    iter_swap(pivot_position, last - upper);// fetch the pivot
    return last - upper; }
  else { iter_swap(pivot_position, begin); return begin; }}
```

Let X be some sorting algorithm requiring at most $n\lg n + cn + o(n)$ comparisons on average. Then, QuickXsort with a Median-of-$\sqrt{n}$ pivot selection also needs at most $n\lg n + cn + o(n)$ comparisons on average. If the unlikely case happens that always the $\sqrt{n}$ smallest elements are chosen for pivot selection, $\Omega(n^{3/2})$ comparisons are performed. However, as the average case results suggest, such a worst case is unlikely. For improving the worst-case complexity, a trick similar to that of Introsort was applied. Choose some small $\delta > 0$. Whenever the pivot

Program 3.2: Implementation of BlockQuicksort.

```
template<template<class , class> class Partitioner, typename iter, typename Compare>
inline void qsort(iter begin, iter end, Compare less) {
  const int depth_limit = 2 * ilogb((double)(end - begin)) + 3;
  iter stack[80]; iter* s = stack;
  int depth_stack[40]; int depth = 0;
  int* d_s_top = depth_stack;
  *s = begin; *(s + 1) = end; s += 2;
  *d_s_top++ = 0;
  do {
    if (depth < depth_limit && end - begin > IS_THRESH) {
      iter pivot = Partitioner< iter, Compare>::partition(begin, end, less);   //Push large
          side to stack and continue on small side
      if (pivot - begin > end - pivot) { *s = begin; *(s + 1) = pivot; begin = pivot + 1; }
      else { *s = pivot + 1; *(s + 1) = end; end = pivot; }
      s += 2; (*d_s_top)++ = ++depth;
    }
    else {
      if (end - begin > IS_THRESH) // if recursion depth limit exceeded
        std::partial_sort(begin, end, end);
      else
        Insertionsort::insertion_sort(begin, end, less); //copy of std::__insertion_sort (GCC
            4.7.2)
      s -= 2; begin = *s; end = *(s + 1); //pop new subarray from stack
      depth = *(--d_s_top);
    }
  } while (s != stack);
}
int main(void) {
  std::vector<int> v = assign_values();
  qsort<Hoare_block_partition_simple>(v.begin(), v.end(), std::less<int>());
}
```

is not contained in the interval $[\delta n, (1-\delta)n]$, choose the next pivot as the median of the whole array (or some linear size sample) using some linear-time worst-case algorithm. This modification results in a sorting algorithm that performs at most $n\lg n + cn + o(n)$ comparisons in the average case and $n\lg n + O(n)$ comparisons in the worst case.

QuickMergesort

One example for QuickXsort is QuickMergesort. For the Mergesort part we use standard (top-down) Mergesort, which can be implemented using m extra element spaces to merge two arrays of length m.

After the partitioning, one part of the array —we assume the first part— has to be sorted with Mergesort. In order to do so, the second half of this first part is sorted recursively with Mergesort while moving the elements to the back of the whole array. The elements from the back of the array are inserted as dummy elements into the first part. Then, the first half of the first part is sorted recursively with Mergesort while being moved to the position of the former second part. Now, at the front of the array, there is enough space (filled with dummy elements) such that the two halves can be merged.

The smaller part of the partitioned array is sorted with Mergesort. Thus, the part which is not sorted by Mergesort always provides enough temporary space. Whenever a data element is moved to or from the temporary space, it is swapped with the dummy element occupying the respective position. Since Mergesort moves the data from left to right, it is always clear which elements are the dummy elements.

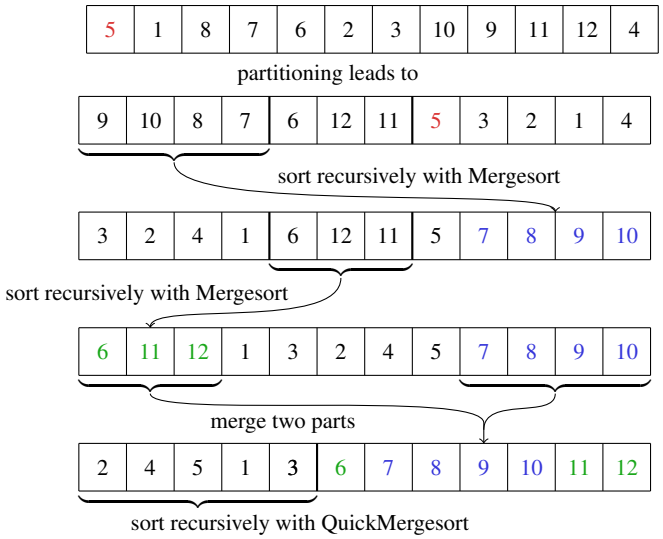


Figure 3.4: Example for the execution of QuickMergesort.

The executed stages of the algorithm QuickMergesort (with no median pivot selection strategy applied) are illustrated in Figure 3.4.

Mergesort requires approximately $n\lg n - 1.26n$ comparisons on average, so that with a median of $\Theta(\sqrt{n})$ elements we obtain an internal sorting algorithm with $n\lg n - 1.26n + o(n)$ comparisons on average. One can do even better if small subarrays are sorted with another algorithm Z requiring less comparisons but extra space and a quadratic number of moves, e. g., Insertionsort or MergeInsertion. When using MergeInsertion as a base, QuickMergesort needs at most $n\lg n - 1.3999n + o(n)$ comparisons and $O(n\lg n)$ other instructions on average. Empirically, MergeInsertion achieved a constant in $[-1.43, -1.41]$ for the *linear* term (for some values of n even below -1.43). The extra space needed is $O(\lg n)$ wor ds for the recursion stack of Mergesort and (depending on the implementation) for MergeInsertion.

Figure 3.5– 3.6 show some results on QuickX-sorting smaller and larger data sets, comparing the number of element comparisons and CPU time of natural competitors.

For the Median-of-3 version called CleverQuickXsort, where the median is chosen of three random elements, it has been shown that QuickMergesort performs at most $n\lg n - 0.75n + o(n)$ comparisons on average.

QuickMergeXsort is QuickMergesort applied to element sets after surpassing of some threshold cardinality X_-THRESH. Below this value, the sorting algorithm X is called and, if X is internal, the sorted elements are moved to their target location expected by QuickMergesort.

Figure 3.3 provides the full implementation details of QuickMerge(X)sort (in C). The realization of sorting algorithm X and the partitioning algorithm (either Lomoto's unidirectional, Hoare's bidirectional, or an alternative partitioner) have to be added. The listing shows that by dropping the base cases from QuickMergesort the code is short enough for textbooks on algorithms and data structures. The general principle is that we have a merging step that takes two sorted areas, merges and swaps them into a third one.

The program *msort* applies Mergesort with X as a stopper. It goes down the recursion tree and shrinks the size of the array accordingly. If the array is small enough, the algorithm calls X followed by a joint movement (memory copy) of array elements (the only change of code wrt QuickMergesort). The algorithm *out* is a first level of recursion of Mergesort – recursive calls are organized differently, thus it serves as an interface between the recursive procedure *msort* and top-level procedure *sort*. It works similarly to the standard Mergesort algorithm. Last, but not least, we have the overall internal sorting algorithm *sort*, which performs the partitioning.

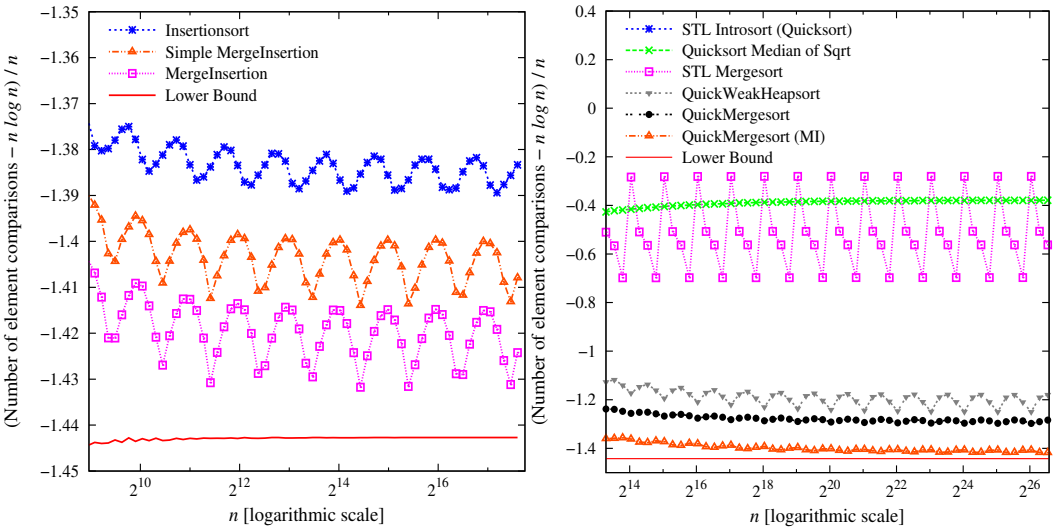


Figure 3.5: Number of element comparisons for sorting with $n \log n + \kappa n$ comparisons for smaller and larger data sets.

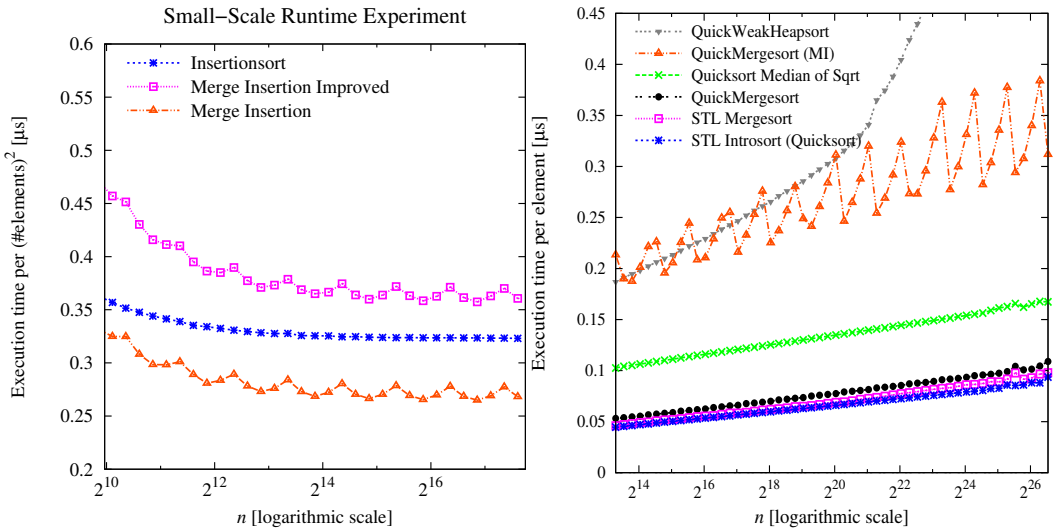


Figure 3.6: CPU time for sorting with $n \log n + \kappa n$ comparisons for smaller and larger data sets.

3.7 Summary

Sorting is at the basis of many algorithms that exploit intelligence. For example the problem of the delayed elimination of duplicates is solved by first sorting the data, and then eliminating subsequent repeated entries. We have seen different contributions to improve sorting by reducing the number of comparison to constant-factor optimality, and by reducing the number of mispredicted branches. The resulting algorithms are theoretically and practically superior to textbook and library implementations.

As Quicksort suffers in an essential way from branch misprediction we presented an approach to address this problem by decoupling control from data flow: in order to perform the partitioning, we have split the input in blocks of constant size; then, all elements in one block were compared with the pivot and the outcomes of the comparisons were stored in a buffer. In a second pass, the respective elements were rearranged. By doing so, we avoided conditional branches based on outcomes of comparisons at all (except for the final Insertionsort).

Program 3.3: Implementation of QuickMergeXsort.

```
typedef std::vector<t>::iterator iter;
void merge(iter begin1, iter end1, iter target, iter endtarget) {
  iter i1 = begin1, i2 = target + (end1 - begin1), ires = target;
  t temp = *target;
  while (i1 != end1 && i2 != endtarget) {
    iter tempit = (*i1 < *i2) ? i1++ : i2++;
    *ires++ = *tempit; *tempit = *ires;
  }
  while(i1 < end1) { *ires++ = *i1; *i1++ = *ires; }
  *(i1 - 1) = temp;
}
void msort(iter begin, iter end, iter target) {
  index n = end - begin;
  if (n < X_THRESH) {
    X(begin, end);
    for(int i=0; i<n; i++) std::swap(begin[i], target[i])
  }
  else {
    index q = n / 2;
    msort(begin + q, end, target + q); msort(begin, begin + q, begin + q);
    merge(begin + q, begin + n , target, target + n);
  }
}
void out(iter begin, iter end, iter temp) {
  index n = end - begin;
  if (n > 1) {
    index q = n / 2, r = n - q;
    msort(begin + q, end, temp); msort(begin, begin + q, begin + r);
    merge(temp , temp + r , begin, end);
  }
}
void sort(std::vector<t> &a) {
  iter begin = a.begin(), end = a.end();
  while (begin < end) {
    iter b = partition(begin,end);
    if (b < begin + (end - begin)/2) { out(begin, b, b+1); begin = b+1; }
    else { out(b+1, end, begin); end = b; }
  }
}
```

3.8 Bibliographic Notes

Heapsort is a sorting algorithm that was developed in the 1960ies by Robert W. Floyd [263] and J. W. J. Williams [657]. The algorithm applies iterated extraction of the smallest (or largest) element, for which an efficient data structure (the heap) is used.

Ultimate Heapsort [384] applies $n \lg n + O(n)$ element comparisons in the worst-case, but the constant factor of the additive linear term is high. Several other variants of Heapsort (e.g., those discussed in [259, 260, 649, 650, 662]) are not worst-case constant-factor optimal with respect to the number of element comparisons performed, or the additive term may be asymptotically higher (e.g. for those discussed in [99, 299, 642, 663, 664]). According to some model assumptions, the average-case number of comparisons in Bottom-Up-Heapsort [649] has been bounded by $n \lg n + O(n)$, whereas Weak-Heapsort [175] leads to at most $n \lg n + 0.1n$ element comparisons in the worst-case, while needing $O(n)$ additional bits.

Introsort has been contributed by [488], while SuperScalarSampleSort is due to [552]. Sedgewick [568] has introduced many improvements to Quicksort. As an example he proposed to switch to Insertionsort (see, e.g.,

[407, Section 5.2.1]). Sedgewick already remarked that choosing the pivots from a larger sample does not provide a significant increase of speed. In the view of the experiments with skewed pivots [380], this is no surprise. The optimal choice of the pivot has been proven by Martinez and Roura in [466].

Dual-pivot Quicksort originates to [666], while Multi-pivot Quicksort has been studied by [26, 27, 432, 655, 656, 465]. QuickMergesort has been proposed by [230], while MergeInsertion, due to Ford and Johnson, is probably best described by D. E. Knuth [407]. *Fine heaps* go back to [101] (and its alternatives to [470, 648]). The stopover optimization method has been proposed by Carlsson [99] (see also [100, 299]). The main loop in BlockQuicksort is based on TunedQuicksort by Elmasry, Katajainen, and Stenmark.

QuickXSort is analyzed in great depth by [233]. It mixes Hoare's Quicksort algorithm with X, where X can be chosen from a wider range of other known sorting algorithms, like Heapsort, Insertionsort and Mergesort. Its major advantage is that QuickXsort can be in-place even if X is not. The work provides general transfer theorems expressing the number of comparisons of QuickXsort in terms of the number of comparisons of X. More specifically, if pivots are chosen as medians of (not too fast) growing size samples, the average number of comparisons of QuickXsort and X differ only by $o(n)$-terms. For median-of-k pivot selection for some constant k, the difference is a linear term. For instance, median-of-three QuickMergesort uses at most $n \lg n - 0.8358n + O(\log n)$ comparisons. Furthermore, the possibility of sorting base cases with some other algorithm using even less comparisons is studied. By doing so, the average-case number of comparisons can be reduced down to $n \lg n - 1.4112n + o(n)$ for a remaining gap of only $0.0315n$ comparisons to the known lower bound (while using only $O(\log n)$ additional space and $O(n \log n)$ time overall). Implementations of these sorting strategies showed that the algorithms challenge well-established library implementations like Musser's Introsort.

There are variants of QuickXsort with improved worst-cases [232], and refinements to MergeInsertion for a better average case [594]. There is a proposal for sorting in optimal complexity of $O(n!) + o(n)$ comparisons [569]. Exact numbers for sorting with $n \leq 22$ have been computed [595]. Performance improvements to BlockQuicksort have been found by [28].

Chapter 4
Deep Learning

The success in learning how to play *Go* at a professional level is based on training a deep neural network on a wider selection of human expert and self-playing games and poses the question on the availability, the limits, and the possibilities of this technique for other combinatorial games, especially, when there is a lack of access to a larger body of additional knowledge.

As a step towards this direction, we train a *value network* for TicTacToe, providing perfect winning information obtained by *retrograde analysis*. Next, we train a *policy network* for the SameGame, a challenging combinatorial puzzle. Here, we discuss the interplay of deep learning with a randomized algorithm for optimizing the outcome of single-player games.

In both cases we observed that ordinary feed-forward neural networks can perform better than convolutional ones in both accuracy and efficiency. For Sokoban we look at imitation and curriculum learning of the search heuristic, as well as a refined design of the loss function in A*.

4.1 Introduction

Deep[1] learning (DL) has been introduced with the objective of moving the field closer to the creation of human-like intelligence. One of its core data structures is a *convolutional neural network* (CNN), a network with a particular connectivity structure among the nodes.

A *neural network* (NN) is a graph representation of a function with input variables in $\mathbb{R}^l$ and output variables in $\mathbb{R}^k$ (for some natural numbers l and k). The internal working is described through an activation function and a threshold applied at each network node. The input vector reflects a number of l features (e.g., in board game the board itself is often included in the feature vector). In a *value network* we have $k = 1$, while in *policy networks* we get a probability distribution for the successor set. Learning is the process of computing weights to the network edges to find a close approximation of the true network function via the (incremental) change of the weights through labeled input/output sample pairs. In *multi-layer feed-forward neural networks* (MLNN) there are input and output, as well as fully connected intermediate hidden layers, while for CNNs the input layers are more loosely connected through several *planes*.

As an example of a simplistic NN take the *perceptron*, a network with no hidden neurons. For input x weighted with w (with weights being added) and activating function ϕ being applied to determine the output by computing $\phi(wx)$. Additionally, we have weighted input for constant 1. The training of a perception for a binary network function is simple: 1) initialize counter i and initial weight vector w_0 to 0; 2) as long as there are vectors for

[1] This chapter is based on joint work with Tristan Cazenave, Leah Chrestien, Tomás Pevný, and Antonín Komenda. It puts together and improves the work from [181, 119, 120].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_4

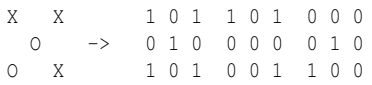


Figure 4.1: A TicTacToe position won for the X player, and its representation in the form of input planes.

which $w_i x \leq 0$ set w_{i+1} to $w_i + x$ and increase i by 1; 3) return w_{i+1}. If the data is linearly separable the algorithm will terminate.

A L-layered MLNN (multi-layered neural network) has layers S_0, S_1, …, S_{L-1}, S_L. Each neuron i in S_l is connected to every j in S_{l+1} with weight w_{ij} (exept 1-neurons). The update is layer-wise and synchronously mixed $x'_j = \phi(\sum_{i \in in(j)} x_i w_{ij})$ with ϕ being differentiable. The task is to find an optimal weight vector that minimizes the network error.

Applications for MLNN include function approximation and classification. It is well-known that all Boolean functions can be computed with a 2-layered MLNN, and continuous real functions and their derivatives can be jointly approximated to an arbitrary precision on a compact set.

For a CNN, there are four building blocks: convolution, non-linearity via the *rectified linear unit* (ReLU), downsampling, and classification. The main purpose of convolution is to extract features from the input. Convolution preserves the spatial relationship of the features by sliding over the input to produce a feature map. Rectification is an operation that is applied per pixel and replaces all negative values by zero. Downsampling reduces the dimensionality of the feature map, e.g, by applying the maximum or average within a neighborhood to avoid overfitting. The last layer often is a fully connected (multi-layer) perceptron for classification, even though support vector machines (SVMs) or other classifiers may also be used.

As in conventional NNs, CNNs are trained with reinforcement learning (backpropagation and stochastic gradient descent). The major advance in learning larger NNs is growing resources in computational power, especially in graphics processing units (GPUs) found on the computer's graphics card.

The aim of this reinforcement technique in game playing is to learn a *policy network*, which outputs a probability distribution of the next move to play. Alternatively, in a *value network*, learning is used to predict the *game-theoretical value* in a single network output node, i.e., its expected outcome assuming perfect play.

The program *AlphaGo* has defeated many top Human Go professionals. It applies a combination of neural network learning given thousands of expert games (in matters of days of GPU computation time) and Monte-Carlo search (see Chapter 5). By the observed playing strength, one gets the impression that the training made the program *understand* the strategic concepts and tactics of Go. *AlphaGo* learned to match the moves of expert players from recorded historical games. Once it had reached a certain degree of proficiency, it was trained further by playing games against other instances of itself. The input was a random permutation of expert game positions, made available in several Boolean input matrices of size 19×19 (some for the occupation and the colors that play, some for further features like liberty). The output was a 19×19 matrix as a predictor for the next move. In other words, the CNN that was trained was a *policy network*.

4.2 Case Study: TicTacToe

We exemplify the learning setup in TicTacToe (Figure 4.1), where we construct and train a *value network*. The game is a classic puzzle that results in a draw in optimal play (this has led movies like *War Games* to use it as an example of a game that consumes unlimited computational power to be solved).

We will use the prominent tensor and optimization programming framework *Torch*, which supports optimizers like stochastic gradient descent as well as several neural network designs and training options. *Tensors* featured by the framework are numerical matrices of (potentially) high dimension. Torch provides an interactive interface

Program 4.1: Finding the winning sets in TicTacToe assuming a perfect hash table for the states.

```
retrograde()
  change = 1
  while (change)
    change = 0
    for each c = 1 .. 5478
        if (solved[c] == UNSOLVED)
          unpack(c)
          succs = successors(moves)
          if (moveX())
            onesuccwon = 0; allsuccslost = 1
            for each i = 1 .. succs
              apply(moves[i],'X')
              onesuccwon |= (solved[pack()] == WON)
              allsuccslost &= (solved[pack()] == LOST)
              apply(moves[i],'_')
            if (succs & onesuccwon)
              solved[c] = WON; change = 1
            if (succs && allsuccslost)
              solved[c] = LOST; change = 1
          else
            onesucclost = 0; allsuccswon = 1
            for each i = 1 .. succs
              apply(moves[i],'O')
              onesucclost |= (solved[pack()] == LOST)
              allsuccswon &= (solved[pack()] == WON)
              apply(moves[i],'_')
            if (succs && onesucclost)
              solved[c] = LOST;
              change = 1
            if (succs && allsuccswon)
              solved[c] = WON;
              change = 1
  for each c = 1 .. 5478
      if (solved[c] == UNSOLVED)
        solved[c] = DRAW
```

for the programming language LUA. For fast execution of tensor operations, it also supports the export of computation to the graphics card (GPU) in CUDA, a GPU programming framework that is semantically close to and finally links to C. The changes to the LUA code for this shift are minimal.

We kick off with generating all 5478 valid TicTacToe positions and determine their true game value by applying *retrograde analysis*, a known backward search technique for constructing strong solutions to games. The according code is shown in Program 4.1.

In one comma-separated value (CSV) *network output* file the accurate results of all positions for training the value network are kept. In the other *network input* file, we record the according intermediate game states. For each position, we take three 3×3 Boolean *planes* to represent the different boards; one for the free cells, one for the X player and one for the O player (see Figure 4.1).

Next, we produce the input and output files for the neural network to be trained and tested. Program 4.2 shows the compilation of entries from the CSV input to the required binary format.

The NN consists of layers that are either fully connected (MLNN) or convoluted (CNN). The according LUA code is shown in Programs 4.3 and 4.4 (top parts). For CNNs it consists of a particular layered structure, which is interconnected through the definition of planes in the form of tensors. The hidden units were automatically generated by the tensor dimensions. This is done through defining submatrices of certain sizes and some additional padding along the border of the planes. After having the input planes T_I represented as tensors and the output planes represented as tensors T_O (in this case a singular value) there are $k-2$ spatial convolutions connected by the tensors $T_I = T_1, \ldots, T_k = T_O$. The information on the size of sub-matrices used and on the padding to the matrix works as follows. All possible sub-matrices of a matrix for a plane (possibly extended with the padding) on both sides of the input are generated. All sub-matrices are pairwise fully interconnected.

Program 4.2: Converting TicTacToe CSV to a tensor file.

```
local Planes = 3
local csvFile = io.open('ttt-input.csv','r')
local input = torch.Tensor(5478,nPlanes,3,3)
local nb = 0
local currentnb = 0
for line in csvFile:lines('*l') do
  nb = nb + 1
  currentnb = currentnb + 1
  local l = line:split(',')
  local plane = 1
  local x = 1
  local y = 1
  for key, val in ipairs(l) do
    input[currentnb][plane][x][y] = val
    y = y + 1
    if y == 4 then
       y = 1
       x = x + 1
    end
    if x == 4 then
       x = 1
       plane = plane + 1
    end
  end
  if currentnb == 5478 then
     currentnb = 0
     nameInputFile = 'ttt-input.dat'
     torch.save (nameInputFile, input)
  end
  if nb == 5478 then
     break
  end
end
csvFile:close()
```

Program 4.3: Learning TicTacToe with an MLNN.

```
require 'nn'
local net = nn.Sequential ()
net:add (nn.Reshape(27))
net:add (nn.Linear(27,512))
net:add (nn.Tanh())
net:add (nn.Linear(512,1))
local nbExamples = 5478
local input = torch.load ('ttt-input.dat')
local output = torch.load ('ttt-output.dat')
dataset = {};
function dataset:size() return nbExamples end
for j = 1, dataset:size() do
  dataset[j] = {input[j], output[j]};
end
criterion = nn.MSECriterion()
trainer = nn.StochasticGradient(net,criterion)
trainer.maxIteration = 1500
trainer.learningRate = 0.00005
trainer:train(dataset)
```

Deep learning in CNNs is very similar to learning in classical NNs. The main exception is an imposed particular network structure and the computational power to train even larger networks to a small error. For global optimization, usually stochastic gradient descent is used.

Programs 4.3 and 4.4 (bottom parts) also show the LUA code for training the network. One can experiment with alternative formulation for the optimization process, but while some experts insist on batched learning to be more effective, often it does not make much of a difference.

Figure 4.2 shows the effect of learning different NN designs given the precomputed classification of all valid TicTacToe positions. We see that larger networks (number of hidden units, HU, number of intermediate planes,

Program 4.4: Learning to play TicTacToe with a CNN.

```
require 'nn'
local nPlanesInput = 3
local net = nn.Sequential ()
local nplanes = 25
net:add (nn.SpatialConvolution(nPlanesInput, nplanes, 3, 3, 1, 1, 0, 0))
net:add (nn.ReLU ())
net:add (nn.SpatialConvolution(nplanes, nplanes, 2, 2, 1, 1, 1, 1))
net:add (nn.ReLU ())
net:add (nn.SpatialConvolution(nplanes, nplanes, 2, 2, 1, 1, 1, 1))
net:add (nn.ReLU ())
net:add (nn.SpatialConvolution (nplanes, 1, 3, 3, 1, 1, 1, 1))
net:add (nn.ReLU ())
print(net)
net:add (nn.Reshape(1*3*3))
net:add (nn.Linear(9,1))
local nbExamples = 5478
local input = torch.load ('ttt-input.dat')
local output = torch.load ('ttt-output.dat')
dataset = {};
function dataset:size() return nbExamples end
for j = 1, dataset:size() do
  dataset[j] = {input[j], output[j]};
end
criterion = nn.MSECriterion()
trainer = nn.StochasticGradient(net,criterion)
trainer.maxIteration = 1500
trainer.learningRate = 0.00005
trainer:train(dataset)
```

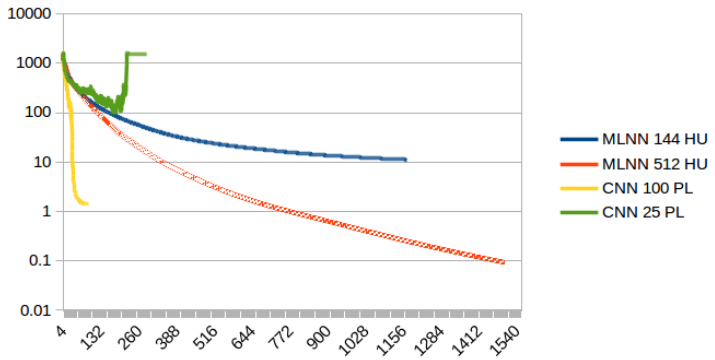


Figure 4.2: Learning results in TicTacToe displaying the training error for the full set with multi-layered neural nets (MLNN) and convolutional neural nets (CNN).

PL) yield better learning. Moreover, CNNs tend to have the smaller number of learning epochs compared to MLNNs. However, each optimization step in a CNN is considerably slower than in an MLNN. While learning MLNNs yields a rather smooth monotone decreasing curve, the learning in a CNN has more irregularities. Moreover, CNNs tend to saturate. Worse, we observe that after reaching some sweet spot CNNs can even deviate back to very bad solutions.

A trained value network can be used as an estimator of game positions (usually called evaluation function) and integrated in other game playing programs. As TicTacToe is known to be a draw we are interested in the accuracy in comparing training with test data.

The following small test shows that the network has learned the value of the game for which we chose 0 for a won game, 50 for a draw, and 100 for a lost game. The value network can be used as an evaluation function and, thus, immediately leads to playing engines. In fact, by fixing the class ranges to 25 and 75, we can use the trained net as an optimal predictor of the true winning classes.

Predicted Value	True Value
95.6280	100
−3.1490	0
50.8897	50
0.6506	0
⋮	⋮

Figure 4.3: Accuracy of value neural network for TicTacToe.

4.3 Case Study: Same Game

The SameGame (see Figure 4.4) is an interactive single-agent game played on an $n \times n$ board with k colored tiles (usually, $n = 15$ and $k = 5$). Tiles are removed if they form a connected group of $l > 1$ elements. Scores sum up, and the *score* of a single move is $(l-2)^2$ points. If a group of tiles is removed, others fall. If a column becomes empty, others move to the left, so that all non-empty columns are aligned. Total clearance yields an additional bonus of 1,000 points. The objective is to maximize the score.

```
4 2 2 5 2 1 5 1 5 5 2 2 1 3 4
4 4 3 1 5 5 2 4 2 3 1 1 5 1 5
1 3 4 5 4 1 4 1 1 4 5 5 2 2 2
3 4 5 1 3 4 1 3 5 5 5 4 1 3 4
2 3 2 4 2 3 1 2 3 2 1 4 5 1 2
1 5 5 4 1 4 5 3 3 3 1 3 4 5 1
3 5 4 5 3 4 2 2 2 4 5 2 1 4 2
2 1 1 5 1 4 2 3 2 1 5 2 4 4 2
2 4 4 3 1 5 4 2 4 1 5 2 1 1 4
1 4 4 5 3 4 1 1 3 2 3 4 5 1 2
1 5 2 3 1 2 4 5 4 4 5 2 5 1 5
3 3 4 2 1 5 1 2 3 5 2 4 4 1 2
4 4 1 3 4 3 2 5 4 2 4 1 3 2 4
2 1 4 3 2 5 5 5 5 1 5 3 2 4 5
2 1 2 1 2 2 3 3 2 1 1 2 5 4 3
```

```
                              4
                              5
                              2
                              4
                              2
                              4
                              2
                            3 5
                          2 1 2
                          3 2 4
1 2       5     3 2 3 5
2 5 1 5 3 5 2 5 2 3
```

Figure 4.4: Initial and final position in the SameGame.

Successor generation has to find the tiles that have the same color. We used an explicit stack for building moves. Termination is checked by looking into each direction for a tile of the same color.

As the access to high-quality expert games in many domains is limited, the question is how to apply (deep) learning in state-space search without the input of human knowledge, where *state-space search* is a general term for the exploration of problem domains to find a solution that optimizes a given cost function. The enumeration of state spaces, however, often suffers from the *state-explosion problem*, which states that the sizes of the spaces are exponential in the number of state variables.

Therefore, we generate input data for training the CNN using a randomized problem solver, being evaluated on a known benchmark set of 20 problem instances with board sizes $n = 15$ (each tile has one of 5 colors.)

The algorithm we choose is Nested Rollout Policy Adaptation (NRPA). It belongs to the wider class of Monte-Carlo search (MCS) algorithms (see Chapter 5), where Monte-Carlo stands as an alias for random program execution. The main concept of MCS is the *playout* (or rollout) of positions, with results in turn changing the likelihood of the generation of successors in subsequent trials. While the algorithm is general and applies to many other applications, we keep the notation close to games and talk about *boards*, *moves*, and *rollouts*. In NRPA every playout starts from an empty board. Two main parameters trade exploitation vs. exploration: the

$\lambda = 0.0005$	$\lambda = 0.005$	$\lambda = 0.05$	$\lambda = 0.5$	$\lambda = 0.2$
0.1429	0.1548	0.1437	0.2084	0.1567
0.1409	0.1418	0.1414	0.2088	0.1537
0.1404	0.1409	0.1398	0.2088	0.1533
0.1400	0.1408	0.1392	0.2088	0.1531
0.1395	0.1408	0.1388	0.2088	0.1527
0.1391	0.1407	0.1384	0.2088	0.1523
0.1387	0.1407	0.1382	0.2088	0.1521
0.1384	0.1406	0.1380	0.2088	0.1519
0.1382	0.1406	0.1378	0.2088	0.1517
0.1380	0.1406	0.1376	0.2088	0.1515
0.1378	0.1405	0.1375	0.2088	0.1515
0.1376	0.1405	0.1374	0.2088	0.1501
0.1374	0.1405	0.1373	0.2088	0.1457
0.1372	0.1404	0.1371	0.2088	0.1423
0.1370	0.1404	0.1349	0.2088	0.1434
0.1368	0.1404	0.1320	0.2088	0.1416
0.1366	0.1403	0.1291	0.2088	0.1373
0.1365	0.1403	0.1327	0.2088	0.1359
0.1363	0.1403	0.1324	0.2088	0.1353
0.1362	0.1402	0.1325	0.2088	0.1348
0.1361	0.1402	0.1323	0.2088	0.1338
0.1360	0.1402	0.1323	0.2088	0.1333
0.1359	0.1401	0.1322	0.2088	0.1344
0.1358	0.1401	0.1322	0.2088	0.1346
0.1357	0.1401	0.1321	0.2088	0.1352
⋮	⋮	⋮	⋮	⋮

Table 4.1: Parameter finding for DL in the SameGame using 1,000 of 33,972 randomly chosen training examples, minimizing the MSE in stochastic gradient descent according to different learning rates λ.

number of *levels* and the branching factor *iteration* of successors in the recursion tree. Successor selection is relative to probabilities for each move which are stored and updated in a vector.

We ran an NRPA(3,100) search 30 times. To cross-compare we also called NRPA(4,100) for each problem. All individual games are recorded, merged, and subsequently split into 33,972 partial states after each move. The partial states are stored in an input file and the move executed is stored in an output file. For training the network all partial states are randomly shuffled to avoid any bias.

To specify a policy network for the SameGame the set of input planes is defined as follows. For each of the five colors plus one for the blank, we define an indicator matrix of size 15×15 for the board, denoting if a tile is present in a cell or not. This amounts to six planes of size 225, so that we have 1,530 binary input features to the neural network. The output tensor file refers to one binary plane of size 15×15 representing a board, with the matrix entries denoting the cells affected by the move. What is learned in this *policy network* is the distribution values of tiles to be selected next.

Table 4.1 shows the effect of varying the learning parameter for the learning process on a fraction of all training examples. In Figure 4.5 we see the effect of learning different neural networks given the precomputed randomly perturbed set of all SameGame training positions. The first iterations of the optimization process are plotted. Again, it seemingly looks like MLNNs perform better in comparison with convolutional structures. Moreover, the convergence was much faster: the largest MLNN took about five hours and the smallest about 1.5 hour, while the CNN took about two days of computational time.

To validate the solution, we compare the MLNN network output after 1,000 learning epochs with the real output. In the visualization of Figure 4.6 we took a threshold of 20% for setting a bit.

More effort will be needed to reduce the error to a value, which can be used for playing well. The subsequent integration of the NN into NRPA, however, is simple, as we only need to change either the initialization or the rollout function. There are three main implementation options: 1) the NN recommendation is used as an initial policy matrix prior to the search. 2) the NN recommendation and the learned policy are alternated by flipping a coin with probability p. 3) the distribution of successors computed by the policy is merged with the NN recommendation.

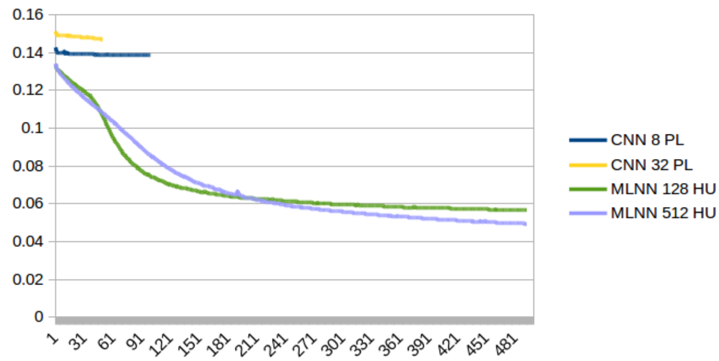


Figure 4.5: Learning results in the SameGame displaying the change of the network error on the full training set of 33972 game positions for multi-layer neural nets (MLNN) and convolutional neural nets (CNN).

```
real output                              predicted output
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0            0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0            0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4.6: Validation of learning result.

4.4 Case Study: Sokoban

This section describes a neural network for planning in grid domains. We introduce the notation and discuss the concrete architecture we used for the Sokoban benchmark domain (see Figure 4.7) We assume to have the grid dimensions h and w to define the network.

The input to the neural network is denoted as $\vec{x} \in \mathbb{R}^{h,w,d_0}$, where h and w is the height and width of the maze respectively, and d_0 varies with the number of channels as explained above. Intermediate outputs are denoted by $\vec{z}^i = L_i(\vec{z}^{i-1})$, where L is some neural network layer (convolution C, attention A, or position encoding E) and for the sake of convenience, we set $\vec{z}^0 = \vec{x}$. All $\vec{z}$ are three dimensional tensors, i.e. $\vec{z}^i \in \mathbb{R}^{h,w,d_i}$. Notice that all intermediate outputs $\vec{z}^i$ have the same width and height as the maze (ensured by padding), while the third dimension, which is the number of output filter(s), differs. Value $\vec{z}^i_{u,v}$ denotes a vector created from $\vec{z}^i$ as $(\vec{z}^i_{u,v,1}, \vec{z}^i_{u,v,2}, \ldots, \vec{z}^i_{u,v,d_i})$. Below, this vector will be called a *hidden vector* at position (u,v) and can be seen as a description of the properties of this position.

Convolution is strictly a local operator. This means that the hidden vector $z^{i+1}_{u,v,\cdot}$ is calculated from hidden vectors $\{z^i_{u',v',\cdot} \mid u' \in \{u-1,u,u+1\}, v' \in \{v-1,v,v+1\}\}$, where we assume convolution to have dimension 3×3. This limits the neural network in synthesizing information from two distant parts of the maze. Yet, we believe that any good heuristic requires such features, since Sokoban is a non-local problem.

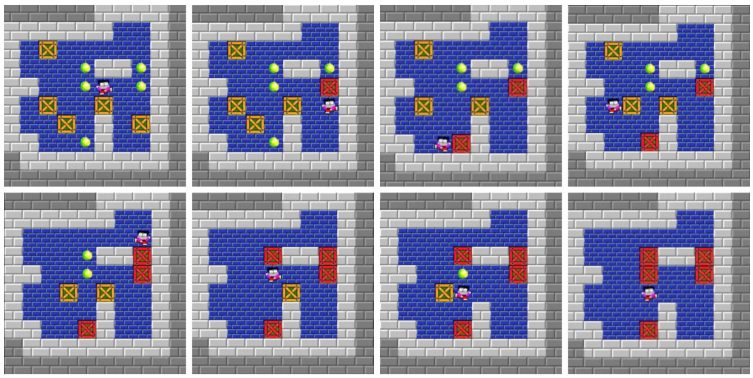


Figure 4.7: A solution to a Sokoban problem.

Therefore, we turn the attention to a self-attention mechanism. The output of self-attention from z^i is calculated in the following manner. At first, the output from previous layer z^i is divided into three tensors of the same height, width, and depth, i.e.

$$\begin{aligned}
\vec{k} &= z^i_{\cdot,\cdot,j} \quad j \in \left\{1,\ldots,\frac{d_i}{3}\right\} \\
\vec{q} &= z^i_{\cdot,\cdot,j} \quad j \in \left\{\frac{d_i}{3}+1,\ldots,\frac{2d_i}{3}\right\} \\
\vec{v} &= z^i_{\cdot,\cdot,j} \quad j \in \left\{\frac{2d_i}{3}+1,\ldots,d_i\right\}
\end{aligned}$$

then, the output z^{i+1} at position (u,v) is calculated as

$$\vec{z}^{i+1}_{u,v} = \sum_{r=1,s=1}^{h,w} \frac{\exp(\vec{q}_{u,v}\cdot\vec{k}_{r,s})}{\sum_{r'=1,s'=1}^{h,w}\exp(\vec{q}_{u,v}\cdot\vec{k}_{r',s'})}\cdot v_{r,s}$$

Self attention, therefore, makes a hidden vector $z^{j+1}_{u,v}$ dependent on all hidden vectors $\{z^j_{r,s} \mid r \in \{1,\ldots,h\}, s \in \{1,\ldots,w\}\}$. The self-attention also preserves the size of the maze. A multi-head variant of self-attention means that z^i is split along the third dimension in multiple $\vec{k}$s, $\vec{q}$s, and $\vec{v}$s. The weighted sum is performed independent of each triple (k,q,z) and the resulting tensors are concatenated along the third dimension.

While self-attention captures information from different parts of the maze, it does not have a sense of a distance. This implies that it cannot distinguish close and far neighborhoods. To address this issue, we add positional encoding, which augments the tensor $z^i \in \mathbb{R}^{h,w,d_i}$ with another tensor $\vec{e} \in \mathbb{R}^{h,w,d_e}$ containing outputs of harmonic functions along the third dimension. Harmonic functions were chosen, because of their linear composability properties. Because the mazes are two dimensional, the distances are split up into row and column distances where $p,q \in [0,d_i/4)$ assigns positions with sine values at even indexes and cosine values at odd indexes. The positional encoding tensor $\vec{e} \in \mathbb{R}^{h,w,d_e}$ has elements equal to

$$\begin{aligned}
\vec{e}_{u,v,2p} &= \sin(\theta(p)u) & \vec{e}_{u,v,2p+1} &= \cos(\theta(p)u) \\
\vec{e}_{u,v,2q+\frac{d_e}{2}} &= \sin(\theta(q)v) & \vec{e}_{u,v,2q+1+\frac{d_e}{2}} &= \cos(\theta(q)v),
\end{aligned}$$

where $\theta(p) = 1/(10000^{\frac{4p}{d_e}})$. On appending this tensor to the input z^i along the third dimension, we get

$$z^{i+1}_{u,v,\cdot} = [z^i_{u,v,\cdot}, e_{u,v,\cdot}].$$

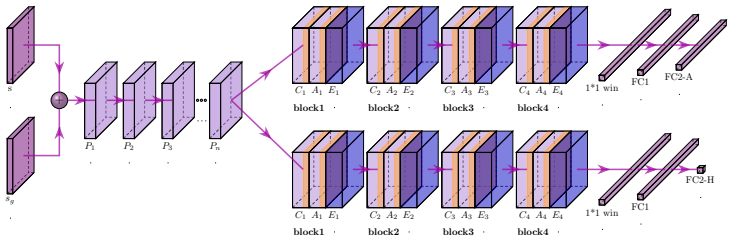


Figure 4.8: Convolutional neural network for grid domains single-agent challenges such as Sokoban.

We use blocks combining Convolution, Attention, and Position encoding, in this order (and call them CoAt blocks), as a part of the NN architecture. The CoAt blocks can, therefore, relate hidden vectors from a local neighborhood through convolution, from a distant part of the maze through attention, and calculate distances between them through position encoding. Since CoAt blocks preserve the size of the maze, they are *scale-free* in the sense that they can be used on a maze of any size. Yet, we need to provide an output of a fixed dimension to estimate the heuristic function and the policy. The output of the last CoAt block is flattened by a 1×1 window, centered around the agent's location and fed to a fully-connected layer and an output head (see Figure 4.8). For example, assuming z^L to be the very last layer, and agent is on position u, v, the vector $z_{u,v,\cdot}$ is of constant dimension equal to the number of channels and is used as an input to the fully-connected layers providing the desired outputs (heuristic values, policy).

Next, we describe the implementation of CoAt blocks in the network architectures we used for all the domains. The network is shown in Figure 4.8. It uses preprocessing convolution layers $P_1, \ldots, P_n$, $n = 7$, (further called *pre-conv*) containing 64 filters where each convolution filter is of the shape 3×3; after the network splits into two heads, it uses four CoAt blocks in each head instead of seven convolution layers. The convolution layers in the CoAt blocks contain 180 filters of size 3×3 each. The attention block uses two attention heads. Each head is finished by two fully-connected layers with reduction to a fixed dimension as described above.

The input to the network is the current state of the game and a goal state, s and s_g, respectively. Each state is represented by a tensor of dimensions equal to width and height (fixed to 10×10 for Sokoban) of the maze $\times$ objects. We used a one-hot encoding of the object states on a grid position (wall, empty, box, agent, and box target), which we could derive automatically. An important design detail is that all convolutions are padded, which means that the output has the same dimension as the input, and they feature skip-connections alleviating a possible vanishing gradient.

We have implemented two different versions of the network (see Figure 4.8) according to their heads: dual-head estimating policy and heuristic value and a single head estimating the heuristic value. In Sokoban, the dual-head representation performed best, while for others a single head estimating heuristic value (no separate head estimating the policy). The presence of more agents would make it difficult to design a policy network in a domain-independent setting and would result in a much larger network, which is inconvenient, time consuming and computationally expensive.

Imitation learning is a framework for learning a behavior policy from demonstrations. We present demonstrations in the form of optimal state-action plans, with each pair indicating the action to take at the state being visited. Generally, imitation learning is useful when it is easier for an expert to demonstrate the desired behaviour rather than to specify a reward function which would generate the same behaviour or to directly learn the policy.

A curriculum refers to an interactive system of instruction and learning with specific goals, contents, strategies, measurement, and resources. The desired outcome of curriculum is a successful transfer and/or development of knowledge, skills, and attitudes. In the context of AI, curriculum learning is a way of training a machine learning model where more difficult aspects of a problem are gradually introduced in such a way that the model is always optimally challenged. Curriculum learning describes a type of learning in which we first start out with only easy examples of a task and then gradually increase the task difficulty. Humans have been learning according to this

principle ever since, but in the common learning setting, we train the neural network on the whole data set. Curriculum learning strategies have been successfully employed in different areas of machine learning, for a wider range of tasks. However, the necessity of finding a way to rank the samples from easy to hard, as well as the right pacing function for introducing more difficult data can limit the usage of the curriculum approaches.

In order to extend the training set without providing any additional plans that the neural network would not be able tosolve, we turn our attention to a form of curriculum learning for neural networks. This approach partially circumvents this problem by re-training from unseen test samples of increasing complexity. In this case, curriculum learning is used to develop scale-free heuristic values for a wider selection of AI planning problems. Specifically, in the experiments, we have quickly reached the capability of planners at larger sizes. To further improve the heuristic function to scale to bigger problems, we re-train the network by extending the training set to include harder problem instances.

We first train the heuristic network on a training set containing easy problem instances quickly solvable by an optimal planner, then use this NN as a heuristic function inside A*, and then extend the training set by more difficult problem instance the NN has solved and finally, re-train the NN. Thus, we perform a bootstrap, where the NN is gradually trained on more difficult problem instances.

This way, curriculum learning plays an important role in improving the performance of the heuristic network on not just the trained dimensions but also on higher dimensions by extrapolation. For curriculum learning, the learning rate is reduced in successive training iterations.

We first describe the details of the training and, then, present the experimental results on Sokoban. The A* algorithms with learnt heuristic functions, realized by the convolution-attention-position networks (further denoted as A*-CoAt), are compared to A* with learned heuristic function realized by convolutional networks (denoted as A*-CNN), and to the planners LAMA, SymBA*, and Mercury. A*-CNN and A*-CoAt uses the vanilla A* search algorithm without any additional tweaks. We also compare the planner to a solution based on reinforcement learning.

On all the compared domains, we analyse the strength of the learnt heuristic and generalization property by solving grid mazes of increasing complexities, approximated by the number of boxes in Sokoban.

The policy-heuristic network we wish to learn accepts a state of a game, s as an input and returns the next action, a, and a heuristic value, $h(s)$, as an output. The training set $\mathscr{X}_{\text{trn}} = \{(s_i, a_i, |\pi^*(s_i)|)\}_{i=1}^{n}$, therefore, consists of $n \approx 10^6$ of these triples, i.e., $\mathscr{X}_{\text{trn}} = \{(s_i, a_i, |\pi^*(s_i)|)\}_{i=1}^{n}$. The triples in the training set were created by randomly generating 40,000 Sokoban instances. Each instance has dimension 10×10 and it always contains only 3 boxes (and an agent). SymBA*, a planner that generates optimal solutions, was used to generate optimal plans π^* for each of these n Sokoban instances. In each plan trajectory, the distance from a current state to the goal state is learned as the heuristic value, $h(s_i)$. Thus, the collection of all state-action-heuristic triples form the training set $\mathscr{X}_{\text{trn}}$.

The Sokoban mazes in the *training set were created with only three boxes*. This means that in the testing set, when we are solving instances with more boxes, we are evaluating its *extrapolation* to more complex unseen environments, which cannot be solved by naive memorisation. However, the limited training set (containing 3 boxes) hinders the full potential of the neural network. With curriculum learning, we fine-tune the neural network using a training set containing Sokoban mazes of dimensions 10×10 with 3 to 7 boxes that have been already solved by the A*-NN with the corresponding architecture. This, therefore, improves the heuristic function without the need to train the network from scratch and, more importantly, without the need to use other planners to create new plans.

The neural network was trained by the *Adam optimizer* with a default learning rate of 0.001 for optimisation. The categorical cross entropy loss function was used to minimise the loss in the action prediction network and the mean absolute error loss was the loss function in the heuristic network. For curriculum learning, the learning rate was reduced to about 1×10^{-4} in successive training iterations. The experiments were conducted in Keras framework with Tensorflow as the backend.

4.4.1 Designing Loss Functions

To study new designs of loss functions we define a search problem instance by a directed weighted graph $\Gamma = \langle \mathscr{S}, \mathscr{E}, w \rangle$, a distinct node $s_0 \in \mathscr{S}$ and a distinct set of nodes $\mathscr{S}^* \subseteq \mathscr{S}$. The nodes $\mathscr{S}$ denote all possible states $s \in \mathscr{S}$ of the underlying transition system representing the graph. The set of edges $\mathscr{E}$ contains all possible transitions $e \in \mathscr{E}$ between the states in the form $e = (s, s')$. $s_0 \in \mathscr{S}$ is the initial state of the problem instance and $\mathscr{S}^* \subseteq \mathscr{S}$ is a set of allowed goal states. Problem instance graph weights (alias action costs) are mappings $w : \mathscr{E} \to \mathbb{R}^{\geq 0}$.

Let $\pi = (e_1, e_2, \ldots, e_l)$, we call π a path (alias a plan) of length l solving a task Γ with s_0 and $\mathscr{S}^*$ iff $\pi = ((s_0, s_1), (s_1, s_2), \ldots, (s_{l-1}, s_l))$ and $s_l \in \mathscr{S}^*$. An optimal path is defined as a minimal cost of a problem instance $\Gamma, s_0, \mathscr{S}^*$ and is denoted as π^* together with its value $f^* = w(\pi^*) = w(e_1) + w(e_2) + \ldots, + w(e_l)$. We often minimize the cost of solution of a problem instance $\Gamma, s_0, \mathscr{S}^*$, namely π^*, together with its length $l^* = |\pi^*|$.

We assume a heuristic to be a function $h_\theta : \mathscr{S} \to \mathbb{R}^{\geq 0}$ mapping a state $s \in \mathscr{S}$ to a real non-negative value, where $\theta \in \mathbb{R}^m$ holds parameters of h. Using a set of problem instances $\mathscr{T}$ (further called training set), we want to optimize parameters θ of h_θ such that a search algorithm (namely, the A* algorithm) would find an (optimal) solution by expanding the least number of states. This, in practice, means to solve optimization problem

$$\arg\min_\theta \sum_{\mathscr{S} \in \mathscr{T}} \mathrm{L}(h_\theta, \mathscr{S}), \tag{4.1}$$

where the optimized function L, further called loss, needs to be designed such that its smaller values imply better heuristic function h_θ as perceived by A*. Needless to say, the L_2 loss does not have this property as discussed later. In the rest of this section, we first state the properties an optimal h_θ for A* should have, and then imprint them in the loss function L.

Let's recall how the A* algorithm works. For consistent heuristics, where $h(s) - h(s') \leq w(s, s')$ for all edges (s, s') in the w-weighted state space graph, it mimics the working of Dijkstra's shortest-path algorithm and maintains the set of generated but not expanded nodes in $\mathscr{O}$ (the Open list) and the set of already expanded nodes in $\mathscr{C}$ (the Closed list). It works as follows.

1. Add the start node to the Open list $\mathscr{O}_0$.
2. Initiate the Closed list to empty, i.e. $\mathscr{C}_0 = \emptyset$.
3. For $i \in 1, \ldots$ until $\mathscr{O}_i \neq \emptyset$
   a. Select the state $s_i = \arg\min_{s \in \mathscr{O}_{i-1}} g(s) + h(s)$
   b. Remove s_i from $\mathscr{O}_{i-1}$, $\mathscr{O}_i = \mathscr{O}_{i-1} \setminus \{s_i\}$
   c. If $s_i \in \mathscr{S}^*$, i.e. it is a goal state, go to 4.
   d. Insert the state s_i to $\mathscr{C}_{i-1}$, $\mathscr{C}_i = \mathscr{C}_{i-1} \cup \{s_i\}$
   e. Expand the state s_i into states s' for which hold $(s_i, s') \in \mathscr{E}$ and for each
      i. if s' is in the Closed list as s_c and $g(s') < g(s_c)$ then s_c is reopened (i.e., moved from the Closed to the Open list), else continue with (e)
      ii. if s' is in the Open list as s_o and $g(s') < g(s_o)$ then s_o is updated (i.e., removed from the Open list and re-added in next step with updated $g(\cdot)$), else continue with (e)
      iii. add s' into the Open list
4. Walk back to retrieve the optimal path.

In the above algorithm, $g(s)$ denotes a function assigning an accumulated cost w for moving from the initial state (s_0) to a given state s. Consistent heuristics are called monotone because the estimated cost of a partial

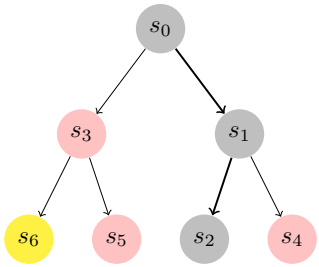


Figure 4.9: A visualization of a search space of an A* algorithm, where sequence of states $s_0 \rightarrow s_1 \rightarrow s_2$ represents the optimal plan, states $\{s_3, s_4, s_5\}/s_6$ are off the optimal path but have / have not been generated by the A*.

solution $f(s) = g(s) + h(s)$ is monotonically non-decreasing along the best path to the goal. More than this, f is monotone on all edges (s, s'), if and only if h is consistent as we have $f(s') = g(s') + h(s') \geq g(s) + w(s, s') + h(s) - w(s, s') = f(s)$ and $h(s) - h(s') = f(s) - g(s) - (f(s') - g(s')) = f(s) - f(s') + w(s, s') \leq w(s, s')$. For the case of consistent heuristics, no reopening (moving back nodes from Closed to Open) is needed, as we essentially traverse a state-space graph with edge weights $w(s, s') + h(s') - h(s) \geq 0$. For the trivial heuristic h_0, we have $h_0(s) = 0$ and for perfect heuristic h^*, we have $f(s) = f^* = g(s) + h^*(s)$ for all nodes s. Both heuristics h_0 and h^* are consistent.

Even if the heuristic is not consistent, algorithms like A* even without the reopening, remain complete i.e. they find a plan if there is one. Plans might not be provably optimal, but are often sufficiently and surprisingly good in planning practice.

In practice, the loss function L is minimized on a training set of problem instances $\mathscr{T}$, but for the sake of brevity, we explain it on a single problem instance $\Gamma = \langle \mathscr{S}, \mathscr{E}, w \rangle$ (the extension to set of plan is trivial through Equation 4.1). We assume to have a (preferably optimal) plan $\pi = ((s_0, s_1), (s_1, s_2), \ldots, (s_{n-1}, s_n))$, and we denote all states from the optimal plan as $\mathscr{S}^o = \{s_0, s_1, s_2, \ldots, s_n\}$. Such a plan can be found by A* with some (admissible) heuristic function h, which *does not have to* coincide with the heuristic function h_θ we are optimizing. We denote states off the optimal plan as $\mathscr{S}^n \subset \mathscr{S} \setminus \mathscr{S}^o$, where the subset exists because, in practice, $\mathscr{S}^n$ contains states generated by A* while solving the problem instance Γ. In the visualization in Figure 4.9, grey states are on the optimal path $\mathscr{S}^o$, pink states are off the optimal path, and yellow states were not generated in the course of solving the problem instance. Hence, $\mathscr{S} = \{s_i\}_{i=1}^{6}$, $\mathscr{S}^o = \{s_i\}_{i=1}^{2}$ and $\mathscr{S}^n = \{s_i\}_{i=3}^{5}$.

4.4.2 Definition of L*

The neural network is used to predict a heuristic for A*. In the limit of the learning process, we want it to minimize the loss function down to value zero, meaning that the training has converged to a heuristic function that perfectly estimates the distance to the goal. For this case, all states on the optimal path would have the same f-value, and all states not on the optimal path would have values that are not smaller. In other words, during the exploration of A* with a perfect heuristic, only nodes s with $f(s) = f^*$ are expanded. Up tie breaking, the optimal heuristic function h_θ forces the A* to expand only states on the optimal path and never non-optimal states. For such a heuristic, we require dominance such that

$$\forall s' \in \mathscr{S}^o) \text{ and } \forall s'' \in \mathscr{S}^n) \text{ we have } g(s') + h_\theta(s') \leq g(s'') + h_\theta(s'') \tag{4.2}$$

On the optimal path, we also want monotonicity, such that

$$\forall s_i, s_j \in \mathscr{S}^o \text{ with } i < j \text{ we have } g(s_i) + h_\theta(s_i) \leq g(s_j) + h_\theta(s_j) \tag{4.3}$$

Notice that in Constraint 4.2, we ignore states that have not been generated by A* simply because they are not in the exploration set. But $\mathscr{S}^n$ will always contain all states of distance one from the optimal path, which is sufficient to show that a loss equalling zero implies expanding states only on an optimal path (in the training set).

To prevent confusion, we emphasize that conditions are designed for the heuristic h_θ that is to be optimized, and not for the heuristic h that has generated the training set in the first place. Wrt. the current exploration during training, the condition does not need hold for a consistent h. Consistency implies that the f-value of all nodes in Closed are smaller-than-or-equal to the ones in Open. Moreover, the function f is non-decreasing on every path that starts in s_0. Encoding these monotonicity requirements for f, however, neither steers the learning process towards better and better heuristic values nor does it converge to the perfect heuristic h^* in the limit, as the trivial heuristic h_0 is also consistent.

While Constraint 4.3 is true for every consistent heuristic, Constraint 4.2 is true only for perfect heuristics. Otherwise, we could have some earlier states in the exploration off the optimal path that have a smaller f-value than later ones on the optimal path. What seems to be over-restrictive, such that almost no heuristic function will ever fulfill, Constraint 4.2, is intentional. We believe it is good to thrive for optimality to obtain something very good in the end.

The loss function L* simply minimizes the number of times each of the above constraints are violated as

$$\begin{aligned}&\frac{1}{|\mathscr{S}^o||\mathscr{S}^n|}\sum_{s'\in\mathscr{S}^o}\sum_{s''\in\mathscr{S}^n}\mathrm{I}\left[g(s')+h_\theta(s')>g(s'')+h_\theta(s'')\right]+\\&\frac{1}{|\mathscr{S}^o|(|\mathscr{S}^o|-1)}\sum_{i=2}^{|\mathscr{S}^o|}\sum_{j=1}^{i}\mathrm{I}\left[g(s_i)+h(s_i)>g(s_j)+h(s_j)\right],\end{aligned}\tag{4.4}$$

where I$[\cdot]$ is an Iverson bracket, which is equal to one if the argument is true and zero otherwise. The first part of the loss function loosely upper bounds the number of non-optimal states the A* expands (see the theorem below), while the second part ensures the monotonicity of the heuristic function along the optimal plan. In other word, Conditions 4.2 and 4.3 encode the aim of a consistent heuristic and the aim for a perfect one. During training, we iterate over many samples of A* explorations, which enlarges the scope of L*.

For setting up constraints for heuristic learning, we only need the partitioning of the set of explored nodes into the sets S^0 and S^n, computed via an optimal plan and a set of all generated nodes, together with their g-values. Given the perfect heuristic, A* will always find an optimal solution. Up to tie-breaking, it is optimal efficient and will expand only nodes with optimal merit f^*.

Loss function L* does not distinguish between the Open and Closed lists in the exploration of A*, as long as it has access to the combined set of explored nodes. This way, we can take any optimal planner and not just heuristic search planners for training. The definition neglects the fact that some nodes in S^n may still reside on an(other) optimal path π'. They may take an active role in processing training sample π'.

Even loss $L^* = 0$ does not imply a perfect heuristic for the entire state space, as the definition is still bound to the exploration sets at plan-finding time. The most we can expect for this case is to prove optimality on the training set. Moreover, perfect heuristics are rare, and we might not be able to converge to $L^* = 0$ in a large problem instance. The experiments will show that the heuristics even for this case, are very effective in reducing the number of expanded nodes, decreasing the plan length, and increasing the coverage. The run time for evaluating a neural net once trained is linear in the size of the network, and much faster to compute than most heuristics in planning.

The evaluation set consists of 2,000 mazes of dimensions 10×10 with 3, 4, 5, 6 or 7 boxes (recall that the training set contain mazes with only 3 boxes). Unless said otherwise, the quality of heuristics is measured by the relative number of solved instances, also known as *coverage*. Table 4.2 shows the coverage of compared planners, where *all* planners were given 10m to solve each Sokoban instance. We see that the classical planners solved all test mazes with three and four boxes but as the number of boxes increase, the A*-NN starts to have an edge. On problem instances with six and seven boxes, A*-CoAt achieved the best performance, even though

				normal		curr.
#b	SymBA*	Mercury	LAMA	CNN	CoAt	CoAt
3	**1**	**1**	**1**	0.92	0.94	0.95
4	**1**	**1**	**1**	0.87	0.91	0.93
5	0.95	0.75	0.89	0.83	0.89	0.91
6	0.69	0.60	0.65	0.69	**0.76**	**0.85**
7	0.45	0.24	0.32	0.58	**0.63**	**0.80**

Table 4.2: Fraction of solved Sokoban mazes (coverage, higher is better) of SymBA*, Mercury, LAMA, A*-CNN (caption CNN) and the A*-CoAt (caption CoAt). A*-CNN and A*-CoAt (with caption normal) use networks trained on mazes with three bozes; A*-CoAt (with caption curr.) used curriculum learning.

#b	SymBA*	Mercury	CNN	CoAt
3	21.40	21.70	24.20	22.20
4	34.00	34.33	40.53	36.00
5	38.82	42.83	45.52	39.11
6	41.11	-	51.00	42.11
7	-	-	54.33	44.17

Table 4.3: Average plan length of SymBA*, Mercury, A*-CNN (denoted as CNN) and that with the A*-CoAt. For clarity, we do not show results of LAMA, as it is performs exactly like SymBA* for 3 and 4 boxes. Column captioned #b indicates the number of boxes in different categories.

it was trained only on mazes with three boxes. The same table shows, that A*-CoAt offers better coverage than A*-CNN, and we can also observe that curriculum learning (see column captioned curr.) significantly improves the coverage.

We attribute SymBA*'s poor performance to its feature of always returning optimal plans while we are content with sub-optimal plans. LAMA had even lower success in solving more complicated mazes than SymBA*, despite having the option to output sub-optimal plans. To conclude, with an increase in the complexity of the mazes, the neural networks outshine the classical planners which makes them a useful alternative in the Sokoban domain.

The average plan length, shown in Table 4.5, reveals that the heuristic learnt by the CoAt network is strong, as the average length of the plans is close to that of SymBA* which always returns optimal solutions. We conclude that the CoAt network delivers a strong heuristic outside its training, much better than that of the CNN network and the planners (for mazes with more than 6 boxes).

Table 4.4 shows the coverage (the percentage of solved mazes) of all compared planners on problem instances with various number of boxes (recall that the NNs were optimized on instances with only three boxes). All planners were given a time limit of 10m to solve each Sokoban instance.

We see that for mazes with 3 and 4 boxes, the optimal planners SymBA* and Delfi were able to optimally solve all problem instances while the best performing architecture among the NNs, which is CoAt optimized with respect to L^* (CoAt-L^*), could solve 94% and 93% of the mazes respectively. Mercury, a satisfying planner that had the option to output sub-optimal plans solves all the mazes containing 3 boxes but performs worse than the NN architectures for mazes with higher complexities. As the number of boxes start to increase, the A^* with NNs start outperforming the classical planners. The same table also shows that A^* with NNs optimizing L^* are consistently better than those optimizing L_2. For example, on the most difficult mazes with seven boxes, CoAt-L^* solves 77% of mazes, CoAt-L_2 solves 63% of mazes and CNN-L_2 solves 51% of mazes.

The average plan length shown in Table 4.5 reveals that the heuristic learnt by the CoAt-L_2 and CoAt-L^* networks is strong, as the average length of the plans is very close to the optimum (that of SymBA* which always returns optimal solutions). Figure 4.10 shows the average number of expanded states of A^* with various versions of heuristics; the trend further proves that CoAt-L^* is the strongest and the quickest as the average number of states is the least among all the NN architectures.

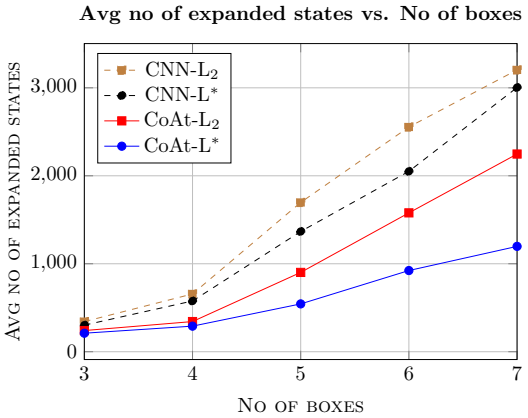


Figure 4.10: Sokoban: The average number of expanded states is the least for CoAt* when trained from only 10000 samples.

				CNN		CoAt	
#*b*	SymBA*	Delfi	Mercury	L_2	L^*	L_2	L^*
3	1	1	1	0.81	0.87	0.91	0.94
4	1	1	0.81	0.74	0.80	0.89	0.93
5	0.94	0.91	0.67	0.72	0.82	0.85	0.89
6	0.55	0.55	0.49	0.61	0.71	0.73	0.80
7	0.46	0.44	0.31	0.51	0.59	0.63	0.77

Table 4.4: Coverage of SymBA*, Delfi, Mercury, CoAt and CoAt* on test data sets containing variable number of boxes. Column captioned #*b* indicates the number of boxes in different categories.

			CNN		CoAt	
n	SymBA*	Mercury	L_2	L^*	L_2	L^*
3	21.40	29.32	30.56	28.67	22.90	22.02
4	34.00	41.00	43.42	41.33	35.11	35.03
5	38.82	45.76	45.34	44.83	40.12	40.12
6	41.11	-	49.82	46.32	42.11	41.65
7	-	-	58.23	56.33	53.33	53.19

Table 4.5: Average plan length of SymBA*, Mercury, A*-CNN (denoted as CNN), A*-CoAt and A*-CoAt*. Column captioned #b indicates the number of boxes in different categories.

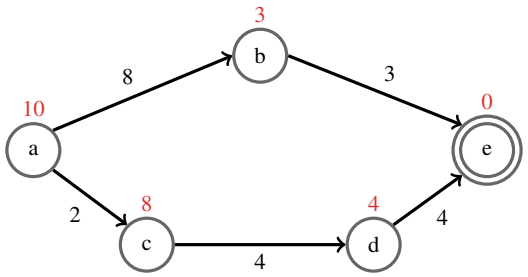


Figure 4.11: Problem instance, where perfect heuristic is not optimally efficient with GBFS. Numbers on the edges denote the cost of action and red numbers next to node denote the minimal cost-to-go.

4.5 Greedy Best-First Search: Optimizing Rank

While the perfect heuristic alias cost-to-goal h^* is the best possible estimate for algorithms like A* (up to tie-breaking) in terms of nodes being expanded, for *greedy best-first search* (GBFS) h^* does not necessarily yield optimal solutions. A* explores all the nodes with $f(s) < f^*(s) = g(s) + h^*(s)$ and some with $f(s) = f^*(s) = g(s) + h^*(s)$, so nodes can only be saved with optimal $f^*(s) = g(s) + h^*(s)$. In fact, when given h* as the heuristic, *only* nodes s with $f(s) = f^*(s)$ are expanded. Depending on the strategy for tie-breaking, the solution path can be found in the minimal number of node expansion, or take significantly longer (e.g., in lower g-value first exploration of the search frontier). Any heuristic other than h^* is either overestimating, and, therefore, may lead to either non-optimal solutions in A*, or weaker than h^*, leading to more nodes being expanded.

Even if h^* is given, GBFS is never guaranteed to be optimal. Consider the following graph with five nodes a,b,c,d,e, weight $w(a,b) = 10$, $w(b,e) = 3$, $w(a,c) = 2$, $w(c,d) = 4$, $w(d,e) = 4$, and $h^*(a) = 10$, $h^*(b) = 3$, $h^*(c) = 8$, $h^*(d) = 4$, $h^*(e) = 0$ (see Figure 4.11), initial node a, goal node e. These numbers are the actual costs and the red numbers are the exact heuristic function. For finding a path from node a to node e GBFS would return (a,b,d) following the heuristic function. However, the path (a,c,d,e) has cost 10 instead of 12.

Therefore, the loss in imitation learning has to be optimized for a corresponding variant of forward search. It is advisable to optimize planning heuristics to rank, not to estimate cost-to-goal. Optimizing the ranking function is a general approach to learn optimal heuristics of search algorithms in deterministic planning. We instantiated it for A* and GBFS searches, leading to two, slightly different loss functions. Optimal rank is a necessary and sufficient condition of optimally efficient heuristic. By linking the optimization problem to statistical learning theory, we observe that learning to rank is easier than regression.

4.6 Summary

Deep learning is very successful in learning images and improving board game play. To reflect its usage in game playing programs this chapter explained the working of learning to train value and policy (neural) networks. In both case studies we excluded human expert knowledge and used accurate and approximate computer exploration results instead.

Of course, deep learning for TicTacToe is shooting small animals with large bullets, especially given that we have computed exact information on the game-theoretical value for all reachable states beforehand. Nonetheless, the learning process was insightful; we were able to train the network to eventually learn the exact winning information in TicTacToe, and —likely due to better separation— the wider the hidden layers, the better the learning.

Next, we turned to the SameGame, for which we applied a fast randomized solver to generate a series of good game. Again, we obtained better learning curves with shallow MLNNs, which also led to a drastic performance gain compared to the CNN designs.

In the deep learning paradigm, the sparser form of convolutions often performs better than ordinary multi-layered neural networks that have fully connected hidden layers. The sparse interconnection between the network levels is balanced by a deeper network by means of a larger number of units adjacent to each other. To some extent the results can be interpreted in the sense that good neural learning does not always have to be *deep*. Depending on the learning task, sometimes *shallow* but *wide* networks are also appropriate.

We also presented general applicable improvements to NN-learning such as self-attraction, imitation, and curriculum learning. They lead to good results in single-agent planning challenges like Sokoban. Adapting the loss function to the search algorithm leads to further advances.

4.7 Bibliographic Notes

Rojas [542] has written a well-respected text book on neural network design and learning. Prototypical applications for convolutional neural networks and deep learning are computer vision, and language understanding. Moreover, deep learning has made its way to playing real-time strategy games by just looking at the screen and score data [478].

As a sign of industrial relevance, Google bought the DL specialist DeepMind. In 2013, DeepMind has impressed with using CNNs to play various Atari 2600 games from the Arcade Learning Environment. In March 2016 (and May 2017), *AlphaGo* won 4:1 (3:0) against Lee Sedol (Ke Jie) in a match; proving itself to be the first computer program to beat top professional human players in Go [576]. The project was led by David Silver, a former PhD student in the Games Group at University of Alberta. Cazenave reproduced *AlphaGo*'s results by training a DNN for the same set of expert games on a GPU-lifted PC. The minimized error evaluation was comparable to the one reported by DeepMind [576]. The accuracy for finding the expert move was 55.6% (*AlphaGo* had a success rate of 57.0%). There is some trickery found for the learning, where a Go-playing program defeats world-class Go AI—but loses to human amateurs [641].

A pioneer in DL was Andrew Ng. He trained large neural networks with millions of images for classification. Today, image recognition by machines trained via deep learning in some scenarios is better than humans, and that ranges from object recognition in uploaded images to identifying indicators for cancer in blood and tumors in MRI scans. With a larger selection of personal data, individualized learning becomes available.

In recent years, DL has become the solution for a further broad range of applications, often outperforming the state of the art. Heuristic search planning with CNNs using imitation, attention and curriculum learning has been considered by [119], and a differentiable loss function for learning heuristics in A* by [120].

Shalev-Shwartz, Shamir and Shammah provide a deeper understanding of the difficulties and limitations associated with common approaches and algorithms [572]. They describe four types of simple problems, for which the gradient-based algorithms commonly used in deep learning either fail or suffer from significant difficulties.

Randomized algorithms often show performance advantages to deterministic algorithms, as in the randomized test for primality [585, 586]. In *Roshambo*, random strategies are superior to deterministic ones. Randomization often is conceptually simple and, frequently, successful in large state-spaces to find the *needle in-the-haystack*. For example, most successful SAT solvers rely on randomized search.

Besides advances in GPU architectures for deep learning (like NVIDIA's Kepler architecture and the P100) Intel Xeon Haswell CPUs also allow the efficient integration of convolutions by taking advantage of SIMD instructions via vectorization and of multiple compute cores via threading [3]. The processors operate on vectors of data up to 256 bits long (8 single-precision numbers) and perform up to two multiply and add operations per cycle. They support vector instruction sets which provide 256-bit floating-point arithmetic primitives, and enhancements for flexible SIMD data movements. These architecture-specific advantages have been implemented, e.g., in the Math Kernel Library and used in deep learning frameworks.

One published NN implementing a heuristic function for Sokoban was proposed by [308]. The network's shape resembled letter Y, as it has two heads, and it contains only convolution layers. The first seven convolution layers were shared (we call them pre-conv layer abbreviating preprocessing-convolution). Then, the network splits to yield two sets of outputs: (i) the estimate of the heuristic function and (ii) the policy. After the split, each path to the output contained seven convolution layers followed by two dense layers. Attention was first introduced in NLP, as it allows to relate distant parts of input together [632]. Imitation learning goes back to work by [517], and curriculum learning to work by [241]. A vanished gradient is discussed by [327]. Planners chosen were LAMA [536], SymBA* [617], Delfi [391] and Mercury [389]. Self-attention has been explained by [622].

Chapter 5
Monte-Carlo Search

The purpose of statistical search algorithms, primarily Monte-Carlo search (MCS), is to make intelligent decisions, even (perhaps especially) in the absence of expert-designed heuristics. It does this by simulating a large number of random actions and using the statistics of the results to refine the decisions as more samples are made. Remarkably, this has been shown to work surprisingly well across a range of problems.

So far MCS has mostly been applied to developing strong game playing agents for challenging games such as Go, where it made a revolutionary step forward. However, there are several properties of the approach, which make it attractive for a range of applications. Across the entire book we extend the number of problems this technique can be applied to.

The origin of MCS are investigations of strategies for solving the multi-arm bandit problem, where, in a casino, different arms of a slot machine can be pulled. The task that leads to trading exploration with exploitation is to establish a close-to-optimal strategy of finding (and pulling) the arm with the best payoff.

In many cases the algorithm requires only a forward model (i.e., given a current system state the forward model gives the next state after taking a particular action) of a system and an objective function that can rate the quality of complete solutions.

5.1 Introduction

Monte-Carlo[1] search (MCS) is a randomized search algorithm which iteratively performs random searches, so-called *rollouts*, within the search space, until the algorithm finds a valid solution, a maximum amount of time has elapsed, or a maximum number of rollouts have been performed. The search method has particularly been applied to solve problems with a huge search space where no adequate lower and upper bounds are available. Improved rollouts perform an additional heuristic that determines next moves within the rollouts, to guide the search.

Nested Monte-Carlo Search (NMCS) extends MCS by the concept of *levels*. For combinatorial search in single-player games it is an apparent alternative to algorithms like UCT that are applied in two-player and general games. To trade exploration with exploitation the randomized search procedure intensifies the search with increasing recursion depth. If a concise mapping from states to actions is available, the integration of policy learning leads to *nested rollout with policy adaptation* (NRPA).

While historically in line, the algorithm NRPA is conceptually different from the other MCS strategies as it does not construct a search, but a recursion tree. In this chapter we will also study methods that improve solution

[1] This chapter is based on joint work with Peter Kissmann, Damian Sulewski, Hartmut Messerschmidt, and Tristan Cazenave. It puts together and improves the work from [211, 187].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_5

diversity: *Beam NRPA* keeps a bounded number of solutions in each recursion level; and *High-Diversity NRPA* further includes refinements that improve the performance of the algorithm.

Monte-Carlo search balances entering unseen areas of the search space (exploration) with working on an already established good solution (exploitation). Many algorithms, however, suffer from a solution process that has many inferior solutions in the beginning of the search. If policies are learnt too quickly, the number of different solutions reduces, and if they are not strong enough, they will not help sufficiently well to enter parts of the search space with good solutions.

In other words, the diversity of the search remains limited. Beam MCS is a combination of memorizing a set of best playouts instead of only one best playout at each level. This set is called a beam and all the positions in the beam are developed.

Beam search carries over to NRPA, enforcing an increased diversity in the set of solutions. In Beam NRPA, for each level of the search, instead of a singleton the algorithm keeps a bounded number of solutions together with their policies in the recursion tree.

5.2 Monte-Carlo Search

The main concept of Monte-Carlo Search (MCS) is the random *playout* or *rollout* of a position, whose outcome, in turn, changes the likelihood of generation successors for subsequent trials. Rollouts were made prominent in applying (temporal-difference) learning to play Backgammon.

5.2.1 Upper Confidence Bounds Applied to Trees

One prominent member in this class of reinforcement learning strategies is *upper confidence bounds applied to trees* (UCT). UCT is a simulation-based method that stores only small parts of the actual game tree.

Essentially, UCT grows a tree and performs rollouts at the leaves. A formula that trades exploration vs. exploitation guides the direction in which the tree should grow (in form of a new leaf). Each round of UCT consists of four steps: *Selection* starting from root, select successive child nodes down to a leaf node. Choosing nodes that lets the game tree expand towards the most promising moves is the essence of MCS; *Expansion*: unless the executed move ends the game, create one or more successor nodes of it and choose from them node v; *Simulation*: play a random rollout from node v; and *Backpropagation*: using the result of the rollout, update information in the nodes on the path from v to the root.

Each simulation run starts at the root node and selects as the next move to take the one maximizing the UCT formula

$$UCT(s,m) = Q(s,m) + C\sqrt{\frac{\log N(s,m)}{N(s)}},$$

with $UCT(s,m)$ being the UCT value of move m in state s, $Q(s,m)$ the average reward achieved when choosing move m in state s, C a constant for controlling the exploration versus exploitation ratio, $N(s)$ the number of times state s was expanded, and $N(s,m)$ the number of times move m was chosen in state s. In single-player games, often the best instead of the average value is back-propagated.

UCT has two phases. In the beginning of each episode it selects actions according to knowledge contained within the search tree. But once it leaves the scope of its search tree it has no knowledge and behaves randomly. Thus, each state in the tree estimates its value by Monte-Carlo simulation. As more information propagates up the tree, the policy improves, and the estimates are based on more accurate returns.

We may ask whether UCT with several runs is better than UCT with only one run. Let us consider the growth $\sqrt{\log(n+k)/n}$ for a fixed value $k = 1$:

$$\begin{aligned}
\sqrt{\log(2)/1} &= 0.83255461115769775634\\
\sqrt{\log(3)/2} &= 0.74115190368375553791\\
\sqrt{\log(4)/3} &= 0.67977799344587264517\\
&\vdots\\
\sqrt{\log(101)/100} &= 0.21482831556480769115\\
\sqrt{\log(1001)/1000} &= 0.08311891950281368148\\
\sqrt{\log(10001)/10000} &= 0.03034870733157594638\\
\sqrt{\log(100001)/100000} &= 0.01072983479132844244\\
\sqrt{\log(1000001)/1000000} &= 0.00371692232336966789\\
\sqrt{\log(10000001)/10000000} &= 0.00126957062627324124\\
\sqrt{\log(100000001)/100000000} &= 0.00042919320537436284
\end{aligned}$$

and the growth of $\sqrt{\log(n+k)/k}$ for a fixed value $k = 1$:

$$\begin{aligned}
\sqrt{\log(2)/1} &= 0.83255461115769775634\\
\sqrt{\log(3)/1} &= 1.04814707396820494648\\
\sqrt{\log(4)/1} &= 1.17741002251547469100\\
&\vdots\\
\sqrt{\log(101)/1} &= 2.14828315564807691178\\
\sqrt{\log(1001)/1} &= 2.62845102281081520349\\
\sqrt{\log(10001)/1} &= 3.03487073315759464562\\
\sqrt{\log(100001)/1} &= 3.39307168579153787872\\
\sqrt{\log(1000001)/1} &= 3.71692232336966844293\\
\sqrt{\log(10000001)/1} &= 4.01473482946985114927\\
\sqrt{\log(100000001)/1} &= 4.29193205374367097775
\end{aligned}$$

All values are small for large n. Moreover, the growth even gets smaller for larger values of k.

Let us now assume an artificial example with a complete binary search tree of depth 30. At all leaves of the left subtree of the root we find a value 10, and in the right subtree all leaves have value 0, except one with the optimum value 100. One random run finds the maximum with probability $1/2^{30}$. Subsequently, we find the sole optimum after an expected number of 2^{30} runs.

UCT, however, will first make two runs, which establishes the value 10 on the left branch with probability 1 and value 0 on the right branch with probability $1 - 1/2^{29}$. Afterwards, the optimum can no longer be achieved, since $\sqrt{\log(n+1)/1} + 10 > \sqrt{\log(n+1)/n} + 0$, so that the right child is never chosen. An increased value of C can close the gap, since $10 + C\sqrt{\log(n+k)/k} < 0 + C\sqrt{\log(n+k)/n}$ implies $\sqrt{\log(n+k)/k} - \sqrt{\log(n+k)/n} > 10/C$. Given that $\sqrt{\log(n+k)/n} \gg \sqrt{\log(n+k)/k}$ we evaluate $\sqrt{\log(n+k)/k} > 10/C$ or $C > 10/\sqrt{\log(n+k)/k}$. Hence, $C = 10/\sqrt{\log(100000001)/1} = 10/4.29193205374367097775 = 2.32$ suf-

fices for $k = 1$, but for $k = 100$ constant C has to be larger as 73,680 to get into the right branch, since $\sqrt{\log(100000100)/1000000000} = 0.00013572281217227264$.

This example indicates that critical decisions at the root are rarely withdrawn, so that drawing more random samples at a node can be advantageous. Next, we briefly look at parallelizations of UCT, achieving a close-to-optimal speedup on multi-core machines.

5.2.2 Parallel Monte-Carlo Search

The first parallelization uses a lock on the variable to increase the number of nodes in the UCT tree. An implementation is shown in Program 5.1. It assumes maximizing the number of moves (as in Morpion Solitaire). The according rollout and backup procedures are displayed in Program 5.2. The main routine initializes the search to the start state and allocates space for the search tree *node* array.

Program 5.1: (Parallel) UCT algorithm.

```
int uct(int depth, int thread) {
  int j = 0, depth = 0, expandleaf[thread] = 1;
  while (node[j].leaf == 0) {
    double maxv = 0; int maxs = -1, succs = 0;
    for (int i=0; i<node[j].numberofsuccs; i++) {
      int s = node[j].successors[i];
      if (node[s].count == 0) {
        maxv = infinity; maxs = s; expandleaf[thread] = 0;
        Succs[thread][succs++] = s; }
      else {
          double v = node[s].value + C * sqrt(log(node[j].count)/node[s].count);
          if (node[s].value == 0) v = 0;
          else if (v > maxv) { maxv = v; maxs = s; }
      }
    }
    if (maxs == -1) { node[j].value = 0; return -1; }
    if (succs > 0) { maxs = next(Succs[thread],succs); }
    node[j].count++; j = maxs; stack[thread][depth] = node[maxs];
    doMove(node[maxs],thread); depth++;
  }
  node[j].count++;
  if (expandleaf[thread]) {
    if (search(sol[thread])) { node[j].value = 0; return -1; }
    else { expandnode(j,thread); insert(sol[thread]); } }
  return j;
}
void uctmontecarlo(int thread) {
  while (1) {
    init(thread);
    index[thread] = uct(depth[thread],thread);
    if (index[thread] > 0) montecarlo(thread);
}
int main() {
  init();
  int highscore = 0;
  for (int i=0;i<THREADS;i++) // spawn light-weight processes
    thread_create(threads[i],uctmontecarlo);
}
```

At a leaf node the UCT tree is enlarged by generating successors of the encountered leaf node. Note that once all successors have been expanded, the selection of successors in the top-down phase of the algorithm is deterministic. If there are still successors left, one is chosen randomly. If a node has been fully explored, its value is set to 0 and is omitted from further processing.

The parallel UCT implementation increases the number of node visits when walking down the tree, applying the UCT formula at every successor. This is different to the sequential implementation where node counts are updated bottom-up. The hope is that different threads likely lead to different leaves.

Program 5.2: (Parallel) Monte-Carlo algorithm.

```
void montecarlo(int thread) {
  int succs = 0;
  while (1) {
    Succs = expand();
    for (i=0; i< Succs.size(); i++)
      if (canMove(Succs[i],thread))
        succ[thread][succs++] = Succs[i];
    if (succs == 0) { break; }
    int r = next(Succs);
    stack[thread][depth[thread]] = succ[thread][r];
    doMove(thread);
    depth[thread]++;
  }
  if (depth[thread] > highscore) highscore = depth[thread];
  int j = index[thread];
  while (j != -1) {
    if (node[j].value < depth[thread])
      node[j].value = depth[thread];
    j = node[j].parent;
  }
}
```

At each node in the Monte-Carlo search, we always generate all successors. If a run is finished, we check, whether a new highscore has been found, so that we can backup the according stack. At the end of the procedure, we store the obtained value at the search tree leaf where the search has started and propagate the outcome bottom-up to the root of the UCT search tree, so that the root value always reflects the optimal value found in its leaves.

An alternative parallelization is to seed the search with a larger set of root nodes and to use UCT to explore the k best of them for a fixed number of nodes.

To achieve this, we use a priority queue, containing the indices of the nodes in the UCT tree along with their corresponding number of expansions, the maximal depth reached, and the resulting UCT value. The queue is ordered according to the UCT values (higher values are better). To come up with a set of states to insert into the queue, first we perform a complete breadth-first search to a certain layer. For each state in this layer, we create one node in the UCT tree (and one corresponding element in the queue).

The removal of the maximal element is done by swapping it with the last one in the queue, decrementing the queue's size and re-establishing the correct order. This can be done in logarithmic worst-case time. We repeat this removal of the maximal element, until we come up with the k best ones.

Typically, we assume $k \gg n$, with n being the number of threads we use. For the parallelization, each thread takes the first unused element and performs a normal UCT run starting at this element. The result will be re-inserted into the queue. For this, at first only the maximal depth and the number of expansions are updated, the corresponding element is swapped with the one at position *size* and the size of the queue is incremented.

At this time, the order of the queue is not correct. As we update the UCT values of all elements in the queue when the total number of expansions changes, we delay this step until all the k nodes are expanded. Then, we update all UCT values and need to reorganize the complete queue.

5.2.3 Nested Monte-Carlo

Nested Monte-Carlo search (NMCS) is a randomized search algorithm that is specifically designed to solve single-player games. Instead of relying on a single rollout at each search tree leaf, the decision-making in level l of the algorithm relies on a level $(l-1)$ search for its successors.

Program 5.3: Nested Monte-Carlo search.

```
double rollout() {
  while (true) {
    if (terminal()) return score();
    int nb = legalMoves(moves);
    if (nb == 0) return 0.0;
    play(moves[rand() % nb]);
  }
}
double nested(int n) {
  Move moves[MaxLegalMoves];
  while (true) {
    int nbMoves = legalMoves(moves);
    if (terminal()) return score();
    for (int i = 0; i < nbMoves; i++) {
      if (n==1) { play(moves[i]); rollout(); }
      else { play(moves[i]); nested(n-1); }
      if (better(score(),bestRollout.score)) update(bestRollout);
    }
    play(bestRollout[n]);
  }
  return 0.0;
}
```

The approach as implemented in Program 5.3 has been successfully applied to solve many challenging combinatorial problems, including Klondike Solitaire, Morpion Solitaire, and SameGame, just to name a few. A large fraction of TSP instances has been solved efficiently at or close to the optimum. NMCS compares well with other heuristic methods that include much more domain-specific information.

NMCS is parameterized with the nestedness *level* of the search which denotes the depth of the recursion. At each leaf of the recursive search a *rollout* is invoked, which performs and evaluates a random run.

5.2.4 Nested Rollout Policy Adaptation

Next, we want to learn a policy within the recursive NMCS procedure. What makes the algorithm different from UCT and NMCS is the concept of learning a policy through an explicit mapping of encoded moves to selection probabilities. As there is no search but a recursion tree, strictly speaking, it is not a tree searching algorithm.

While the NMCS investigates all possible moves in depth $d = l$ of the decision tree in level $l-1$, Nested Rollout Policy Adaptation (NRPA) executes n nested searches in level $l-1$, that all start at the root of the decision tree and follow a policy until they reach a leaf. After a search in level $l-1$ has been performed, the results are evaluated at level l and the policy is updated by the best solution currently found. The algorithm helped finding a new world record in *morpion solitaire*, and high-quality solutions in *crossword puzzles*. As we will see, NRPA is a general search procedure and applies to many games as well as practical applications.

As the set of available successors is finite, a random selection based on the current policy is applied. This biased roulette-wheel links to fitness selection in genetic algorithms. If a successor has been determined, the solution is extended by one step, and the playout loop continues finding the next one.

Figure 5.1 depicts the recursion tree of NRPA with two levels and four iterations. It illustrates the effects of higher and lower-level searches on the level policies. Thus, a policy is either progressed, adapted, or copied.

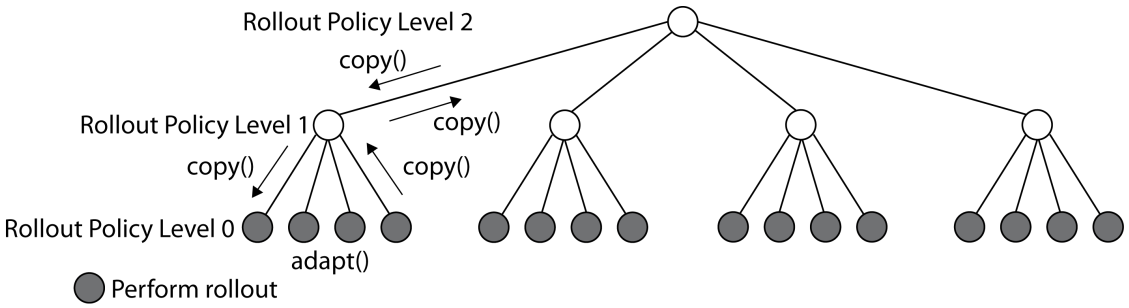


Figure 5.1: NRPA with two levels and four iterations.

The pseudo-code of the recursive search procedure is shown in Algorithm 5.1. NRPA has two main parameters that trade exploitation with exploration: the number of levels l and the branching factor N of successors in the recursion tree. To further accelerate the search, an initial policy can be provided.

Algorithm 5.1: Nested rollout with policy adaptation.

```
procedure NRPA(level l, policy p)
begin
    if l = 0 then
        Best ← Playout(p)
    else
        p'_l ← p
        Best ← (Init,⟨⟩)
        for i = 1,...,N
            r ← NRPA(l − 1, p'_l)
            if r better than best then
                Best ← r
                Adapt(best, p, p'_l)
    return Best
end
```

If we impose further constraints on the TSP such as capacity restriction or the satisfaction of time windows for visiting the cities, finding optimal solutions is difficult already for moderate values of n. For the TSP with time windows (TSPTW) Program 5.4 applies NRPA to learn valid and short tours. The knowledge about which city is the best successor for a given one is trained in the policy, which is adapted for each incoming good tour.

The change of values are as follows. The policy values for successor cities in a good solution are incremented, while the others get an equal share of an according decrement.

Program 5.4: Solving the problem of the traveling salesman with time windows.

```
Tour search(int level) {
  Tour best = new Tour();
  best.score = MAXVALUE;
  if (level == 0) {
    rollout();
    best.score = evaluate();
    for (int j = 0; j < N+1; j++) best.tour[j] = tour[j];
  } else {
    for (int k = 0; k < N; k++)
      for (int n = 0; n < N; n++)
        local[level][k][n] = policy[k][n];
    for (int i = 0; i < ITERATIONS; i++) {
      Tour r = search(level - 1);
      double score = r.score;
      if (score < best.score) {
        best.score = score;
        for (int j = 0; j < N+1; j++) best.tour[j] = r.tour[j];
        if (level > 2)
          System.out.println("Level : " + level + ", score : " + score);
        adapt(best.tour,level);
      }
    }
    for (int k = 0; k < N; k++)
      for (int n = 0; n < N; n++)
        policy[k][n] = local[level][k][n];
  }
  return best;
}
```

Program 5.5 finds the legal moves and evaluates a state. The accounting of constraint violation is used in the overall cost function. The rollout and policy adaptation methods are shown in Program 5.6.

5.3 Beam NRPA

Beam NRPA is an extension of NRPA that maintains B instead of one best solution in each level of the recursion. The motivation behind Beam NRPA is to warrant search progress by an increased diversity of existing solutions to prevent the algorithm from getting stuck in local optima.

The basic implementation of the Beam NRPA algorithm is shown in Algorithm 5.2. Each solution is stored together with its score and the policy that was used to generate it. Better solutions are inserted into a list, which is kept sorted with respect to the objective to be optimized.

As the NRPA recursion otherwise remains the same, the number of playouts to a search with level L and (iteration) width N rises from N^L to $(N \cdot B)^L$. To control the size of the beam, we allow different beam widths B_l in each level l of the tree (common values for B_l are $(1, 10, 10, 10)$). At the end of the procedure, B_l best solutions together with their scores and policies are returned to the next higher recursion level. For each level l of the search, one may also allow the user to specify a varying iteration width N_l. This yields $\prod_{l=1}^{L} N_l B_l$ rollouts for the algorithm Beam NRPA.

Program 5.5: TSP with time windows: evaluation and legal moves.

```
public double evaluate() {
  double makespan = 0.0;
  double cost = 0.0;
  int prev = 0; // starts at the depot
  int violations = 0;
  for (int i = 1; i < N; i++) {
    int node = tour[i];
    cost += dist[prev][node];
    makespan = Math.max(makespan + dist[prev][node], left[node]);
    if (makespan > right[node]) violations++;
    prev = node;
  }
  cost += dist[prev][0];
  makespan = Math.max(makespan + dist[prev][0], left[0]);
  if (makespan > right[0]) violations++;
  return 100000.0 * violations + cost;
}
public int legalMoves() {
  int prev = 0; // starts at the depot
  int opindex = 0;
  double makespan = 0.0;
  double cost = 0.0;
  for (int i = 1; i < tourSize; i++) { // computes makespan
    int node = tour [i];
    cost += dist[prev][node];
    makespan = Math.max(makespan + dist[prev][node], left[node]);
    prev = node;
  }
  if (tourSize > 0)
    prev = tour[tourSize - 1];
  for (int i = 1; i < N; i++) {
    if (!visited[i]) {
      moves[opindex] = i;
      boolean tooLate = false;
      for (int j = 1; j < N; j++)
        if (j != i && !visited[j])
            if ((makespan <= right[j]) && (makespan + dist[prev][j] <= right[j])
                && (Math.max(makespan + dist[prev][i],left[i]) > right[j])) {
              tooLate = true;
              break;
            }
      if (!tooLate)
        opindex++;
    }
  }
  if (opindex == 0)
    for (int i = 1; i < N; i++)
      if (!visited[i])
        moves[opindex++] = i;
  return opindex;
}
```

5.4 Refinements

In the following we look at several algorithmic refinements to Beam NRPA.

Program 5.6: TSP with time windows: rollout and policy adaptation.

```
    public void rollout() {
      int node = 0;
      tourSize = 1;
      for (int i=1;i<N;i++) visited[i] = false;
      while (tourSize < N) {
        int successors = legalMoves();
        for (int i = 0; i < successors; i++)
          value[i] = Math.exp(policy[node][moves[i]]);
        double sum = value[0];
        for (int i = 1; i < successors; i++)
          sum += value[i];
        double mrand = random.nextDouble() * sum;
        int i = 0;
        sum = value[0];
        while (sum < mrand) {
          i++;
          sum += value[i];
        }
        tour[tourSize++] = moves[i];
        visited[moves[i]] = true;
        node = moves[i];
       }
       tour[tourSize++] = 0; // Finish at the depot;
    }
    void adapt(int[] tour_param, int level) {
      for (int k = 1; k < N; k++)
        visited[k] = false;
      int successors;
      int node = 0;
      for (int ply = 0; ply < N; ply++) {
        successors = 0;
        for (int i = 1; i < N; i++) {
          if (!visited[i]) {
            moves[successors] = i;
            successors++;
          }
        }
        local[level][node][tour[ply]] += 1.0;
        double z = 0.0;
        for (int i = 0; i < successors; i++)
          z += Math.exp(policy[node][moves[i]]);
        for (int i = 0; i < successors; i++)
          local[level][node][moves[i]] -= Math.exp(policy[node][moves[i]]) / z;
        node = tour[ply];
        visited[node] = true;
      }
    }
}
```

First, we observe that copying the policy in each rollout of Beam NRPA is a rather expensive operation that can dominate the runtime of the entire algorithm.

In fact, further code analysis showed that the policy update is always performed wrt. the currently best solution found in a level and the policy one level up, so that it is not required to store the policy attached each solution, as long as we keep B_l best policies alive for each level l of the recursive search procedure.

For a faster processing of policy adaptation, we avoid the regeneration of successors by providing all the information that is needed at the time we construct the solution in the rollout. Hence, we store the sequence of codes

Algorithm 5.2: Beam nested rollout with policy adaptation.

```
procedure Beam-NRPA(level l, policy p)
begin
    if l = 0 then
        (score, rollout) ← Playout(p)
        return (score, rollout, p)
    Beam_l ← (Init, ⟨⟩, p)
    for i = 1, ..., N
        L ← ∅
        for t ← (score, rollout, p) ∈ Beam
            L ← L ∪ {t}
            T ← Beam-NRPA(l − 1, p)
            for t' ← (score', rollout', p') ∈ T then
                Adapt(rollout', p', p)
                L ← L ∪ {t'}
        Beam_l ← B best in Beam_l ∪ L
    return Beam
end
```

$Code_l$ and successor node codes $Succ_l$ for each best solution (relative to a level l) produced, where the *code* is a user-specified domain-specific address in the policy table, calculated for the current state and the current move executed in this state.

The implementation in Algorithm 5.3 shows that this strategy is already applicable to the original NRPA algorithm. It leads to minor extensions to the implementation of the generic *playout* function: each time a successor is checked for availability the corresponding code is stored.

Algorithm 5.3: A variant of policy adaptation for NRPA that refers to stored data.

```
procedure NRPA-Adapt(level l, policy p, policy p')
begin
    for c_i ∈ Code_l
        p'[c_i] ← p'[c_i] + α
    z ← 0
    for c' ∈ Succ_{l,i}
        z ← z + exp(p[c'])
    for c' ∈ Succ_{l,i}
        p'[c'] ← p'[c'] − α · exp(p[c'])/z
end
```

We see that the update in *Adapt* affects only the codes of the good solution to be adapted and its successor codes, to balance the positive effect put on choosing it as negative effect to all of its successors.

To avoid fragmented access to the memory and operating system calls to provide memory, high-speed algorithm implementations often avoid dynamic memory allocation or have their own memory maintenance and allocators.

Beam NRPA pre-allocates the information in the beam in static arrays and operates on the stored information directly. Besides faster insertion and deletion this allows one to follow the progress of the search by showing the top $k \leq B_l$ elements.

5.5 Improving the Diversity

Beam NRPA itself is inspired by the objective of higher diversity in the solution space of NRPA. In larger search spaces NRPA often got stuck with inferior solutions. It simply takes too long to backtrack to less determined policies to visit other parts in the search space.

The beam is maintained in a bounded number of *buckets*. The information stored in the buckets of a beam is visualized in Figure 5.2. Instead of the moves executed in a rollout we store the *Code* of the chosen move and the code of its successors *Succ*. Additionally, the length of the rollout and its score are stored for each bucket in the beam.

5.5.1 Improving Diversity in the NRPA Driver

When looking at a beam, a natural question is to warrant that the solutions kept in the beam are substantially different. This can be imposed by matching the best obtained rollout with the ones stored in the beam. Duplicate solutions are excluded from the beam. Algorithm 5.4 provides a pseudo-code implementation.

The application of a filter to improve diversity is implemented in method *Similar*. We expect that $s_i = s_j$ implies $Similar(s_i, s_j)$ and $Similar(s_i, s_j) = Similar(s_j, s_i)$. The output is a truth value (interpreted as a number in $\{0, 1\}$). The beam is scanned for similar states, and if present, the new insertion request is rejected. Such similarity can be implemented on top of the score of the solution, the solution length, or other features of the rollout. The example implementation in Algorithm 5.5 looks at the score and the length of the rollout.

The concept of similarity implies a formal characterization of *solution diversity*. Let $\mathscr{S}$ be a set of solutions of an optimization problem with $B = |\mathscr{S}|$, and let $Similar(s_i, s_j) \in [0, 1]$ be a similarity score between every pair of solutions s_i and s_j, $1 \leq i, j \leq B$, where $Similar(s_i, s_j) = 0$ if $s_i = s_j$. The *diversity* is the sum of pairwise similarities, i.e., $diversity(\mathscr{S}) = \sum_{s_i, s_j \in \mathscr{S}, i \neq j} Similar(s_i, s_j)$. This means that if the solutions in $\mathscr{S}$ are pairwise similar, the diversity in $\mathscr{S}$ is low. A similar concept is that of pre-sortedness in an input array by adding the pairwise number of inversions. Some algorithms can adapt to a varying degree of pre-sortedness.

One important aspect is that adaptation is now applied in every iteration, while before it was applied only for improved solutions. This increases the number of calls significantly but allows more information to be passed between the members in the beam. If the parameters are chosen carefully, the efforts for the playouts and for executing policy adaptation are roughly the same.

We also skip some θ_l iterations before we start learning. The motivating objective is the *secretary problem*, in which the best secretary out of n rankable applicants should be hired for a position. Applicants are interviewed one after the other and the final decision must be made immediately after the interview. The optimal stopping rule rejects the first n/e applicants after the interview and then stops at the first applicant, who is better than every applicant interviewed so far.

Diversity is an objective that must be dealt with care. In some domains the solution length already is the score, so that only solutions of different lengths are kept in the beam. This may limit the number of good solutions in the beam (too) drastically. As a solution to this problem, one could include other state features into the fractional part of the solution.

A good compromise is needed. Using the entire state vector for similarity detection requires comparing regenerated solutions, which can be slow, or storing the full state in the rollout to be retrieved in later calls of the policy adaptation, which would result in a significant overhead in space and time.

Algorithm 5.4: Beam NRPA with high diversity.

```
procedure HD-NRPA(level l, policy p)
begin
    for b = 1,...,B_l
        score_{l,b} ← Init
    if l = 0 then
        Score_{0,1} ← Playout(p)
        return Score_{0,1}
    for i = 1,...,N_l
        score ← HD-NRPA(l − 1, p)
        if score better than Score_{l,B_l} then
            for b' = 1,...,B_{l−1} then
                if ¬ Similar(Score_{l,b'}, Length_{l,b'}, l)
                and Score_{l−1,b'} better than Score_{l,B_l} then
                    insert (Score_{l−1,b'}, Length_{l−1,b'},
                            Code_{l−1,b'}, Succ_{l−1,b'}) into Beam_l
        if (i > Θ) then
            HD-Adapt (l, p'_l)
    return Score_{l,1}
end
```

.

Algorithm 5.5: Example of applied similarity measure.

```
procedure Similar(score s, length r, level l)
begin
    for b = 1,...,B_l
        if Score_{l,b} = s and Length_{l,b} = r then
            return true
    return false
end
```

.

5.5.2 *Improving Diversity in the Policy Adaptation*

We refine the beam search by a reduction of elements eligible to be included in the beam. Therefore, we use $(c_j, c_i) \in Beam_{l,1..b-1}$ to denote that the best rollout code (defined by (c_j, c_i)) in a given level is already present in the prefix of the beam to bucket b in level l. This avoids overly stressing good solutions that have already influenced the policy to be learnt. We also do not want to update elements twice. The according code is shown in Algorithm 5.6. The main function *HD-Adapt* calls the function *HD-Other*, which works as a filter and collects the codes of moves that should be used to change the policy.

We used simple arrays for the data structure to check that a code and set of successor codes is contained in the beam and thus learnt already. Profiling revealed that a significant part of the running time is spent here. Surely, a hash map would be more efficient for checking $(c_j, c_i) \in Beam_{l,1..b-1}$. However, the algorithm has to be modified as the hash map then has to support deletion, given that elements in the buckets being dominated by incoming solutions are removed from the beam, and, thus, no longer serve for duplicate detection in the form of membership queries.

Given that the selection strategy of the successors does not prune away moves that are required to generate an optimal rollout sequence, NRPA and Beam NRPA are *probabilistically complete* in the sense that an optimal solution will eventually be found.

For applying high-diversity candidate solution selection, this theoretical assertion is not necessarily preserved. However, the technique preserves non-zero probabilities for generating the optimal solution in the rollout function. Given that the selection strategy of the successors does not prune away moves that are required to generate an optimal rollout sequence, *high diversity nested rollout with policy adaptation* is probabilistically complete in

Algorithm 5.6: Policy adaptation within HD-NRPA.

```
procedure HD-Other(level l, index b, i, j)
begin
    L ← ∅
    for c_j ∈ Succ_{l,b,i}
        if (c_j, c_i) ∉ Beam_{l,1..b−1} then
            L ← L ∪ {c_j}
    for b' = b+1, ..., B_l, c_{i'} ∈ Code_{l,b'}
        if c_i = c_{i'} then
            for c_{j'} ∈ Succ_{l,b',i'}
                if c_{j'} ∉ L ∧ (c_{j'}, c_i) ∉ Beam_{l,1..b−1} then
                    L ← L ∪ {j'}
    return L
end

procedure HD-Adapt(level l, policy p, policy p')
begin
    p' ← p
    for b ∈ 1, ..., B
        for c_i ∈ Code_{l,b}
            if c_i ∉ Beam_{l,1..b−1} then
                p'[c_i] ← p[c_i] + α
                L ← HD-Other(l, b, i, j)
        z ← 0
        for c ∈ L
            z ← z + exp(p[c])
        for c ∈ L
            p'[c] ← p'[c] − α · exp(p[c])/z
end
```

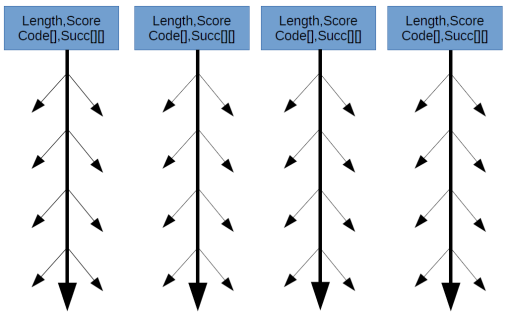


Figure 5.2: Information stored in HD-NRPA; buckets stand for the beam, thin arrows indicate successors (codes, stored in *Succ*), the thick arrow the best solution (codes, stored in *Code*).

the sense so that an optimal solution will eventually be found. This, however, does not imply any performance quality like the ε-optimality of the resulting search algorithms.

5.6 Case Study: SameGame

The SameGame (Figure 4.4) has already been introduced as an interactive game frequently played on hand-held devices. It is solvable in polynomial time for one column of tiles but NP-complete for two or more columns and five or more colors of tiles, or five or more columns and three or more colors of tiles.

Algorithm 5.7: Counting tiles of color c present at index i.

```
procedure Count(index i, color c)
begin
    if Outside(i, j) ∨ color[i] ≠ c ∨ seen[i]
        return 0
    seen[i] ← 1
    return 1+ Count(i+1,c) + Count(i−1,c)+Count(i+n,c)+ Count(i−n,c)
end
```

Algorithm 5.8: Generating and executing moves together with termination criterion in the *SameGame*.

```
procedure LegalMove(Move moves[], length l)
begin
    succs ← 0
    seen ← (0..0)
    if only one move for tabu color then
        tabu ← blank
    for i = 0..n^2 − 1
        if color[i] ≠ blank then
            if ¬seen[i] then
                moves[succs] ← buildMove(i)
                if |moves[succs]| > 1 then
                    if color[i] = tabu then
                        if |moves[succs]| ≤ 2 ∧ l > 10
                            succs ← succs + 1
                    else succs ← succs + 1
    if succs = 0 then
        for i = 0..n^2 − 1
            if color[i] ∉ {blank,tabu} then
                if ¬seen[i] then
                    moves[succs] ← buildMove(i)
                    succs ← succs + |moves[succs]| > 1
    if succs = 0 then
        seen ← (0..0)
        for i = 0..n^2 − 1
            if color[i] ≠ blank then
                if ¬seen[i] then
                    moves[succs] ← buildMove(i)
                    succs ← succ + |moves[succs]| > 1)
    return succs
end
```

```
procedure Play(Move move)
begin
    Sort(move)
    for i in 0..|move| − 1
        Remove(i)
    c ← 0
    for i = 0..n − 1
        if column c is empty
            RemoveColumn(c)
        else
            c ← c + 1
    Score ← Score + (|move| − 2)^2
    if board is empty
        Score ← Score + 1000
    end
end
```

```
procedure Terminal()
begin
    for i = 0..n^2 − 1
        if color[i] ≠ blank then
            if possibleMove(i) then
                return 0
    return 1
end
```

Successor generation and evaluation of the score have to reach out for the tiles that have the same color. Algorithm 5.7 illustrates a recursive implementation for counting the number of successors. To assist the compiler, in the tuned implementation of the SameGame we use an explicit stack for building the moves. Termination can be checked faster by testing each of the four directions of every tile location for having the same color.

Let us briefly look at the efficiency of the implementation (Algorithm 5.8). Let n^2 be the board's total number of cells and t be the number of tiles for a given move. The critical aspect is the adjustment of the board according to the gravity of tiles. For each tile removed, we *bubble* the blank upwards.

- *LegalMoves*, once applied for each state: The construction of one move, including all stack operations, is proportional to the size t of each color group $O(t)$, by virtue of recording tiles visiting for one state, the total of generating all successors is $O(n^2)$.
- *Play*, once applied for each state: besides the removal, executing a move is proportional to t. Removal of a tile group with adjustment in one column (one for each move): $O(mt)$, for an amortized total of $O(n^3)$ in the playout, as at most n^2 tiles can be removed. Removal of one column (selective execution): $O(n^2)$, but

the work amortizes: at most n columns can be removed in the playout. Prior to the removal we sort the tiles affected in each move, which in total (since $N = N_1 + \ldots + N_k$ implies $\sum_{i=1}^{k} N_i \lg N_i = O(N \lg N)$) is bounded by $O(n^2 \lg n^2) = O(n^2 \lg n)$.

- *Terminate*, once applied for each rollout: $O(n^2)$.

The selection of successors is based on the computation of legal moves and the roulette rule selection, where the latter is dominated by the former. If $n \times n$ is the board's dimensions of the SameGame, the time for generating a solution of length l in one playout is bounded by $O(ln^2 + n^3)$.

Table 5.1 compares the scores of one level 4 (iteration 100) HD-NRPA and 30 $\times$ level 3 (iteration 100) HD-NRPA searches both obtained with beam width 10 and initial offset for learning 10. The results are cross-compared with standard NRPA and NMCS.

ID	NMCS(4)	NRPA(4)	HD-NRPA(4)	max. 30 runs HD-NRPA(3)
1	3,121	**3,179**	3,145	3,133
2	3,813	**3,985**	**3,985**	3,969
3	3,085	3,635	**3,937**	3,663
4	3,697	**3,913**	3,879	3,887
5	4,055	4,309	**4,319**	4,287
6	4,459	**4,809**	4,697	4,663
7	**2,949**	2,651	2,795	2,819
8	**3,999**	3,879	3,967	3,921
9	4,695	4,807	**4,813**	4,811
10	**3,223**	2,831	3,219	2,959
11	3,147	3,317	**3,395**	3,211
12	3,201	3,315	**3,559**	3,461
13	3,197	**3,399**	3,159	3,115
14	2,799	3,097	**3,107**	3,091
15	3,677	3,559	**3,761**	3,423
16	4,979	5,025	**5,307**	5,005
17	4,919	**5,043**	4,983	4,881
18	5,201	5,407	**5,429**	5,353
19	4,883	5,065	**5,163**	5,101
20	4,835	4,805	5,087	**5,199**
Sum	77,934	80,030	**81,706**	74,753

Table 5.1: Results in the SameGame.

We can see that improving the diversity generally gives better results than NMCS and NRPA, even though, through randomization, there are problem instances where the opposite is true.

5.7 Case Study: Snake-in-the-Box

The *snake-in-the-box* problem is a longest path problem in a d-dimensional hypercube. Long snakes have an impact on the generation of improved error-correcting codes. During the game the snake increases in length but must not approach any of its previous visited vertices with Hamming distance 1 or less.

The HD-NRPA implementation applies bit manipulation to integers in $0..2^d - 1$. The information on snake visits is kept in a perfect hash table of size 2^d. One optimal solution of length 50 for $d = 7$ is as follows: 0, 1, 33, 35, 43, 42, 10, 26, 27, 25, 57, 56, 48, 52, 53, 55, 63, 62, 126, 122, 123, 115, 113, 81, 80, 88, 92, 93, 95, 87, 86, 22, 6, 7, 15, 13, 12, 44, 108, 104, 105, 73, 75, 67, 66, 98, 102, 103, 101, 69, 68.

There are known generalizations to the problem. First, instead of having a Hamming distance of at least $k = 2$ for the incrementally growing head to all previous nodes of the snake (except the ones preceding the head), one may impose a minimal Hamming distance $k > 2$ to all previous nodes (inducing a Hamming sphere that must not be revisited). In Figure 5.2 (left) we give the best-known solution lengths for the (k,n) snake problem, where

k/d	2	3	4	5	6	7
3	4*v	3*v	3*v	3*v	3*v	3*v
4	7*v	5*v	4*v	4*v	4*v	4*v
5	13*v	7*v	6*v	5*v	5*v	5*v
6	26*v	13*v	8*v	7*v	6*v	6*v
7	50*v	21*v	11*v	9*v	8*v	7*v
8	98*(95)	35*v	19*v	11*v	10*v	9*v
9	190	63(55)	28*v	19*v	12*v	11*v
10	370	103	47*(46)	25*v	15*v	13*v
11	707	157	68	39*v	25*v	15*v
12	1,302	286	104	56(54)	33*v	25*v
13	2,520	493	181	79	47(46)	31v

k/d	2	3	4	5	6	7
3	6*v	6*v	6*v	6*v	6*v	6*v
4	8*v	8*v	8*v	8*v	8*v	8*v
5	14*v	10*v	10*v	10*v	10*v	10*v
6	26*v	16*v	12*v	12*v	12*	12*v
7	48*v	24*v	14*v	14*v	14*	14*v
8	96*(92)	36*v	22*v	16*v	16*	16*v
9	188	64(55)	30*v	24v*	18*v	18*v
10	358	102	46*v	28v*	20*v	20*v
11	668	160	70(64)	40v*	30*v	22*v
12	1,276	288	102	60(56)	36*v	32*v
13	2,468	494	182	80	50*v	36*v

Table 5.2: Best known results in snake-/coils-in-the-box, validated with HD-NRPA.

an asterisk (*) denotes that the optimal solution is known. The validation of the results in generating a solution with HD-NRPA that matches the given bound is indicated with a *v*. For the first problem not solved, the best solutions are shown in brackets.

There is another variant, which asks for a closed cycle, by means that the snake additionally has to bite its own tail at the end of its journey. The algorithm's implementation must take care that this is in fact possible. In Figure 5.2 (right) we give the best-known solutions lengths and the validation results. In summary, using HD-NRPA we could validate all but three optimal solutions in the snake- and coils-in-the-box problems. Approximate solution lengths for the first unsolved problem are shown in brackets.

5.8 Case Study: Vehicle Routing

In the *vehicle routing problem* (VRP) we are given a fleet of vehicles, a depot, and a time delay matrix for the pairwise travel between the customers' locations, service times, time windows and capacity constraints; the task is to find a minimized number of vehicles with a minimized total distance that satisfies all the constraints. Clearly, by choosing only one vehicle, VRP extends the capacitated traveling salesman with time windows. We chose instances of the Solomon VRPTW benchmark for the experiments, a well-studied selection of 100-city problem instances.

VRPs in practice are complex. For example, instead of the straight-line distances, shortest paths for a road network have to be precomputed, leaving a distance matrix to be forwarded to the VRP solver. Often concurrent pickups while delivering items to customers are requested, which has an immediate effect on the violation of capacity constraints. Similarly, for courier express services, items are collected at one site and brought to another. Additionally, there are same-day delivery requirements and reglementations of drivers breaks. The point we stress is that all of these additional constraints can be added into a VRP solver based on random playouts like NRPA, as it incrementally generates a tour.

Program 5.7: Vehicle routing: rollout including evaluation.

```
private double rollout() {
  for (int j=1;j<N;j++) visits[j] = 1;
  visits[0] = V-1;
  tour[0] = 0;
  tourSize = 1; // start node already visited
  int node = 0, prev = 0;
  double makespan = 0.0, capacity = 0.0;
  violations = 0;
  vehicles = V;
  double cost = -1.0 * sumservice;
  while(tourSize < N+V-1) {
    double sum = 0.0;
    int successors = 0;
    for(int i = 0; i < N; i++) {
      if (check(i) && d[node][i] != 1000000.0) {
        moves[successors++] = i;
        for (int j = 0;j < N;j++)
          if (i != j)
            if (check(j))
              if (l[i] > r[j] || makespan + d[node][i] > r[j]) {
                successors--;
                break;
              }
      }
    }
    if (successors == 0) {
      for (int i = 0; i < N; i++)
       if (check(i)) moves[successors++] = i;
    }
    for(int i=0; i<successors; i++) {
      value[i] = Math.exp(global[node][moves[i]]);
      sum += value[i];
    }
    double mrand = random.nextDouble()*sum;
    int i=0;
    sum = value[0];
    while(sum<mrand) sum += value[++i];
    prev = node; node = moves[i];
    tour[tourSize++] = node;
    visits[node]--;
    cost += d[prev][node];
    makespan = Math.max(makespan + d[prev][node],l[node]);
    capacity += w[node];
    if (node == 0) {
      if (prev == 0) vehicles--;
      prev = 0;
      makespan = capacity = 0.0;
    }
    if (capacity > max_capacity) violations += 1.0;
    if (makespan > r[node]) violations += 1.0;
  }
  tour[tourSize++] = 0;
  cost += d[node][0];
  if (node == 0) vehicles--;
  makespan = Math.max(makespan + d[node][0], l[0]);
  if (makespan > r[0]) violations += 1.0;
  return (100000.0 * vehicles) + 100000.0 * violations + cost);
}
```

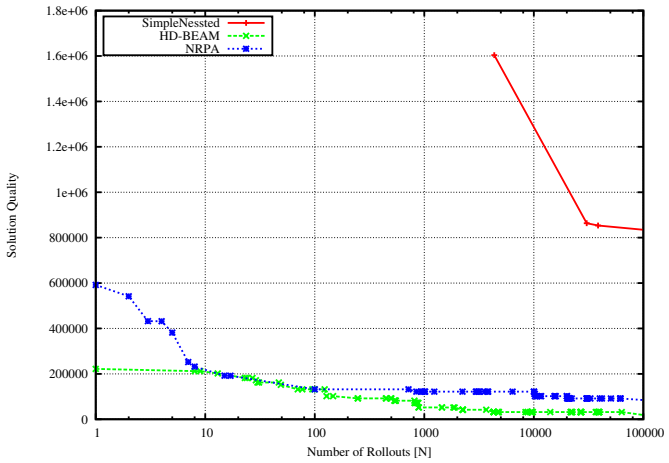


Figure 5.3: Comparing the learning in VRP of Nested MCS, NRPA, and HD-NRPA.

The implementation of the problem is based on the simple observation that a tour with V vehicles can be generated by a single vehicle, where the time (makespan) and the capacity of the vehicle are reset at each visit to the depot. A possible rollout function in Java is shown in Program 5.7. It also tries to reduce the number of vehicles.

In contrast to all other cities the depot is allowed to be visited more times. In this implementation the i-th visit to the depot gets the ID i and must be revisited. The solver has the selective strategy that whenever a candidate city invalidates reaching another city it is discarded from the successor set. We selected a (level 5, iteration 50) search with threshold zero to start learning.

Figure 5.3 compares the different single-agent Monte Carlo search processes for the first 100 thousand playouts of the r101 problem. We see that HD-NRPA shows the fastest learning progress.

5.9 Summary

Monte-Carlo search is class of random search algorithms that has led to a paradigm shift in AI game playing from enumeration to randomization. We have looked at bandit-based search as a reinforcement learning algorithm that applies random playouts. In UCT's tree descent we selected the node that optimizes solution quantity, analogous to the multiarmed bandit problem in which a player must choose the slot machine (bandit) that maximises the estimated reward each turn.

Moreover, nested Monte-Carlo search and nested rollout policy adaptation have shown to be viable options to solve hard combinatorial problems, combining random exploration with learning.

Besides looking at parallel UCT implementations, we added more diversity to the NRPA search, making it faster in several domains. A few implementation refinements and a more careful handling of the solutions stored in the beam made the algorithm perform convincingly across three domains. For the SameGame we exemplified the interface with the generic solver and analyzed the complexity of one playout. Besides elaborating on the given setting and its impact, for the eager algorithm engineer, we also provide pseudo-code implementations.

5.10 Bibliographic Notes

With the success of applying reinforcement learning to play expert-level Backgammon [605], the concept of sampling the outcome of a game in random playouts has been around. Later, bandit-based Monte-Carlo planning and UCT [408] extended the use of playouts and changed the way in which computer play many two-player and general games. The history of playing games with an increasing level of MCS performance is long, with the possible climax in *AlphaGo* defeating a professional Go player in a match.

Around 2006 Rémi Coulomb and other researchers provided a new approach to move planning in computer Go, now known as MCS. Kocsis and Szepesvári (2006) formalized this approach into the UCT algorithm. Actually, their publication studied more general MDP (Markov Decision Process) problems.

Parallelizing UCT on multiple cores is a hot research topic. Different approaches with lock-free hash tables and tree and root parallelizations have been proposed [243, 107], all with individual pros and cons.

If the number of visits is small, the UCT mechanism gives unreliable results. The *rapid action-value estimator* (RAVE) combines UCT with the all-move-as-first heuristic [286] *Prior knowledge* [285] is an alternative for biasing the selection strategy with heuristic knowledge. If an evaluation function is available, with *implicit minimax* MCS makes use of it for a potential gain in performance [435]. *Progressive widening* [113, 138] consists of reducing the move set artificially when the selection strategy is applied and increasing its size progressively (given enough time). When the number of visits of a node equals the threshold, progressive widening prunes most of the children. To get more out of the simulations they should mimic intelligent play. Simulation strategies aim to have more realistic and therefore informative playouts. Domain-independent strategies include Last-Good-Reply [32], MAST [54], which exploits that moves good in one position are likely to be good in other positions, too, and *N-grams*, which keep track of move sequences instead of single moves [607].

MCS is the state of the art in playing two-player games such as Go or Hex [354] or in puzzles, like the Pancake problem [280], and has been applied also to other problems than games, like mixed-integer programming, constraint problems, mathematical expression, function approximation, physics simulation, cooperative pathfinding, as well as planning and scheduling [74]. A comprehensive survey on Monte-Carlo search (MCS) algorithms has been given by [86].

Nested rollouts were introduced by [665]. In further applications, this heuristic is improved successively to apply the algorithm for solving challenging combinatorial problems such as Klondike Solitaire [53].

Cazenave [105] has invented *nested Monte-Carlo search* (NMCS), a randomized search algorithm inspired by UCT [408]. Besides playing games, the NMCS algorithm solves mathematical problems [73].

As a randomized search procedure NRPA has been very successful in solving a variety of optimization problems, including puzzles [545], but also hard optimization tasks in logistics like *constraint traveling salesman problems* [195], combined *pickup-and-delivery tasks* [193], *vehicle routing* [281], and *container packing* [197] problems.

Solving complex TSPs is generally dominated by methods from Operations Research [514, 490], including optimal approaches using Mixed-Integer (Linear) Programming, Branch-and-Bound, and Branch-and-Cut, as well as suboptimal approaches using Large Neighborhood Search, Particle Swarm Optimization, Genetic or Ant Algorithms, Simulated Annealing, etc. [441, 574].

For a growing board SameGame is known to be hard [49]. The formal definition of the Snake-in-the-Box problem and its variants as well as heuristic search techniques for solving it are studied by [502]. The randomized beam search algorithm has been parallelized and applied to solve instances of *Morpion Solitaire* [535]. In Beam NRPA [109], the combination of Beam search and NRPA has been studied.

Part II
Big Data

Chapter 6
Graph Data

The use of digital data has increased rapidly in the last few years. Traditional information sources like books, pictures, letters, or vinyls have been substituted by ebooks, digital photography, eMails, video, and audio. *Big data* is a wording that has been chosen to emphasize that this change is accompaigned by a rising amount of stored information. While the physical space is reduced, the information carried remains the same. Therefore, the selection, analysis, and evaluation of these amounts of data are important.

Big data like hypertexts, computer infrastructures, or social networks are best represented as graphs in which certain non-trivial relations are to be established. Therefore, in this chapter we solve fundamental NP-hard graph optimization problems like Maximum Clique and Minimum Graph Coloring.

As there are many different problem variants to look at, we are interested in general but efficient solver prototypes. The optimization problems are implemented as single-agent games in a search framework, with very little problem-specific knowledge.

As one solving technology we employ Nested Monte-Carlo Search (with and without rollout policy adaptaption) and compare the computational results with UCT and with satisfiability (SAT) solver technology.

6.1 Introduction

Graphs[1] are everywhere. Hypertexts like Wikipedia are steadily expanding. At the very far end we have the Internet itself as a huge arrangement of linked pages. As one indicator, the envisioned set of IP addresses has been extended. Computer networks also grow on a large scale; we have mobile devices and more and more intelligent computational objects, a trend which is sometimes denoted as the *Internet of Things*.

Studies on social networks receive a lot of attention from both sociology and computer science. A social network is a social structure made up of a set of vertices for social actors (such as individuals or organizations) and a set of edges that serve as communication links between these actors. Social network analyses have been done in many aspects of sociology, such as social influence, social groupings, inequality, disease propagation and communication of information. However, a larger amount of data and resources makes it critical to analyze social network-related problems to find optimal solutions.

Many of the first problems shown to be NP-complete were graph problems. Among them we find Graph Coloring (a.k.a. Chromatic Number), Clique, Independent Set, Vertex Cover (Directed/Undirected), Hamilton Circuit, and Hitting Set. Most of these decision problems inherit natural optimization criteria.

Inspired by its initial successes, besides other optimization options we choose Nested Monte-Carlo Search for solving these problems in a search *framework* that links a (domain-specific) combinatorial problem to a

[1] This chapter is based on joint work with Eike Externest, Sebastian Kühl, and Sabine Kuske. It improves the work from [191].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_6

(domain-independent) search algorithm. For Graph Coloring, we add more functionality like finding cliques to initialize the coloring process with a prior, a selective policy to accelerate the search, as well as Greedy Search.

6.2 Mathematical Encoding

Graph Coloring was the first fundamental problem where a computer proved a central result (namely that each planar graph can be colored in four colors; finding it is still NP-hard!).

More formally, the Graph Coloring Problem asks for a vertex coloring with at most k colors, where a *coloring* is a function that maps each node in the graph to a color, so that no two vertices attached to an edge share the same colors.

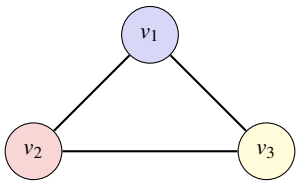


Figure 6.1: Example graph with a 3-Coloring.

Input: Graph $G = (V, E)$, natural number $k \leq |V|$.
Output: 1, if a mapping $c : V \to \{1, 2, \ldots, k\}$ exists with $c(v) \neq c(v')$ for all $\{v, v'\} \in E$; 0, otherwise.

The optimization problem asks for the minimal number of colors, i.e., the smallest possible k that preserves the above condition. This number is called *chromatic number* of G and is denoted by $\chi(G)$. Graph coloring is a problem in many areas of computer science, e.g., for computing timetables.

6.3 PDDL Encodings

All the above graph search problems have a natural representation in the problem domain description language (PDDL). For example, Programs 6.1 and 6.2 provide the PDDL representation of Clique in the form of a domain and problem instance file. In the implementation we count the number of nodes in the clique and for each node selection we delete all the nodes that are not adjacent to it from further consideration. Instead of optimizing the size of the clique in the metric, we provided the decision variant in the form of an incrementally adjusted upper bound to the planner.

We ran initial experiments with an off-the-shelf planner and solved the problem displayed in Programs 6.1 and 6.2 instantly. When updating k from 3 to 4, the planner correctly reports *no plan*.

To test larger problem instances, we wrote a parser that transforms DIMCAS benchmarks into PDDL. However, even for the simplest problem instance in the benchmark set the planner got stuck. Changing the search algorithm and the parameter setting made no essential difference. We concluded that PDDL action planners have difficulties in solving hard combinatorial graph problems.

Program 6.3 provides a PDDL implementation for Graph Coloring. For Independent Set, Vertex Cover, or Hitting Set, the PDDL encodings look similar.

Program 6.1: PDDL domain for Clique.

```
(define (domain clique)
(:requirements :fluents :typing :conditional-effects)
(:types node)
(:predicates
  (adjacent ?n1 ?n2 - node)
  (visited ?n - node)
)
(:functions (size))
(:action select
:parameters (?m - node)
:precondition (and (not (visited ?m)))
:effect
    (and
      (increase (size) 1)
      (visited ?m)
      (forall (?n - node)
        (when (and (not (visited ?n))
                   (not (adjacent ?m ?n)))
              (visited ?n))))))
```

Program 6.2: PDDL instance for Clique.

```
(define (problem three)
(:domain clique)
(:objects n6 n5 n4 n3 n2 n1 - node)
(:init
  (adjacent n1 n2) (adjacent n2 n1)
  (adjacent n2 n3) (adjacent n3 n2)
  (adjacent n3 n5) (adjacent n5 n3)
  (adjacent n1 n6) (adjacent n6 n1)
  (adjacent n6 n4) (adjacent n4 n6)
  (adjacent n4 n5) (adjacent n5 n4)
  (adjacent n5 n6) (adjacent n6 n5)
  (= (size) 0)
)
(:goal (and (>= (size) 3))))
```

6.4 SAT Encodings

To decide whether or not an undirected graph $G = (V,E)$ can be colored with k colors, for each node $v_i \in V$ we introduce indicator variables $x_{i,1}, x_{i,2}, \ldots, x_{i,k}$, where $x_{i,j}$, $i \in \{1,2,\ldots,|V|\}$ and $j \in \{1,2,\ldots,k\}$, denote that node v_i is assigned to color j. We impose that each node is colored;

$$f_1 = \bigwedge_{v_i \in V} \bigvee_{1 \leq j \leq k} x_{i,j},$$

and that no two nodes connected via one edge share the same color:

$$f_2 = \bigwedge_{\{v_h, v_i\} \in E} \bigwedge_{1 \leq j \leq k} (\neg x_{h,j} \vee \neg x_{i,j}).$$

The formula $f = f_1 \wedge f_2$ is satisfiable if and only if the graph enjoys a valid k-coloring.

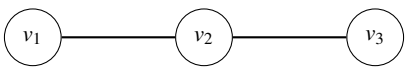


Figure 6.2: Example graph for finding a 2-Coloring.

Program 6.3: PDDL encoding for Graph Coloring.

```
(define (domain coloring)
(:requirements :fluents :typing :conditional-effects)
(:types col node)
(:predicates
  (adjacent ?n1 ?n2 - node)
  (used ?n - node)
  (color ?n - node ?c - col)
)
(:functions (colors))
(:action select
:parameters (?m - node ?c - col)
:precondition (and
  (not (used ?m))
  (forall (?n - node)
  (or
    (not (adjacent ?n ?m))
    (not (used ?n))
    (not (color ?n ?c))))
)
:effect (and
  (used ?m)
  (when (forall (?n - node) (not (color ?n ?c)))
    (increase (colors) 1))
  (color ?m ?c))))
```

As an example take $G = (V,E)$ with $V = \{v_1, v_2, v_3\}$, $E = \{\{v_1, v_2\}, \{v_2, v_3\}\}$ and $k = 2$, shown in Figure 6.2. We have $f_1 = (x_{1,1} \vee x_{1,2}) \wedge (x_{2,1} \vee x_{2,2}) \wedge (x_{3,1} \vee x_{3,2})$ and $f_2 = (\neg x_{1,1} \vee \neg x_{2,1}) \wedge (\neg x_{1,2} \vee \neg x_{2,2}) \wedge (\neg x_{2,1} \vee \neg x_{3,1}) \wedge (\neg x_{2,2} \vee \neg x_{3,2})$. The satisfying assignments $x_{1,2}$, $x_{2,1}$, $x_{3,2}$, and $x_{1,1}$, $x_{2,2}$, $x_{3,1}$ correspond to the following two colorings: 1) $color(v_1) = 2$, $color(v_2) = 1$ and $color(v_3) = 2$; 2) $color(v_1) = 1$, $color(v_2) = 2$ and $color(v_3) = 1$.

For the other graph problems, the SAT encodings are similar.

The procedure by Davis, Putnam (1960) and by Davis, Logemann, Loveland (1962) is the basis of most modern SAT solvers. It combines backtracking with constraint propagation (unit propagation).

The Davis-Putnam-Loveland-Longmann (DPLL) algorithm shown in Programs 6.4 and 6.5 tries finding an assignment $a \in \{0,1\}^n$ for a Boolean function $f : \{0,1\}^n \to \{0,1\}$ in conjunctive normal form (CNF), so that $f(a) = 1$. It solves the satisfiability problem (SAT) problem, which is the first problem proven to be NP-complete.

6.5 Game Encodings

In the encoding of Graph Coloring as a single-player game the player starts at an arbitrary graph node and chooses in each step a next node until all nodes are selected. The result is a permutation of the graph nodes used as an input for the greedy coloring algorithm. The smaller the color number found by the algorithm the higher is the score of the play.

High-Diversity NPRA (HD-NRPA) elaborates on this observation to increases the diversity of the search and provides further algorithmic advances.

The graph optimization problems we consider are cast as single-agent games so that the interface adapts the nomenclature of a board game. The framework supports Monte-Carlo search algorithms like NMCS, NRPA and HD-NRPA, together with different implementations for the policy table.

A move (play) corresponds to a selection of graph nodes. They are stored in the rollout which is bound by a Boolean condition (terminal). The length and score are recorded, and the score is either minimized or maximized (one global flag). Finding the potential set of successors (legalMoves) finalizes the implementation.

Program 6.4: DPLL driver and solver.

```
public void solve(int nClauses, String[][] literals) {
    ArrayList<Clause> Clauses = new ArrayList<Clause>();
    for(int i=0;i<nClauses;i++) {
        Clause clause = new Clause();
        for(int j=0;j<literals[i].length;j++)
            clause.addLiteral(literals[i][j]);
        Clauses.add(clause);
    }
    System.out.println((!DLL(Clauses) ? "UN" : "") + "SAT.");
}
boolean DLL(ArrayList<Clause> Clauses) {
    while (true) {
        String literal = searchSingleLiteral(Clauses);
        if (!literal.equals("UNIT")) {
            removeClauses(literal,Clauses);
            cutClauses(literal,Clauses);
            if (Clauses.size() == 0) return true;
            if (False(Clauses) || Empty(Clauses)) return false;
        }
        else break;
    }
    ArrayList<Clause> C1 = new ArrayList<Clause>(), C2 = new ArrayList<Clause>();
    for (Clause c: Clauses) {
        Clause d = new Clause();
        for (String s: c.literals) d.addLiteral(s);
        C1.add(d);
    }
    for (Clause c: Clauses) {
        Clause d = new Clause();
        for (String s: c.literals) d.addLiteral(s);
        d.add(d);
    }
    Clause cl1 = new Clause(), cl2 = new Clause();
    String l1 = pickLiteral(Clauses), l2 = "";
    if (l1.startsWith("-")) l2 = l1.substring(1); else l2 = "-"+l1;
    cl1.addLiteral(l1); cl2.addLiteral(l2); C1.add(cl1); C2.add(cl2);
    return DLL(C1) || DLL(C2);
}
```

If not provided a priori, in NRPA each rollout (calling the constructor) is initially empty. Initially, the input is read from the DIMACS files and stored in an adjacency matrix (an adjacency list implementation is also be available). Output information on the search process and the improvement of the solution qualities is reported on the screen.

Program 6.6 shows the running code for Clique, which is is the same for Independent Set (invert graph) and Vertex Cover (invert the graph and set $k' := |V| - k$). In the variation denoted by $+$ in the experiments we additionally maintain a list *open*, so that only nodes adjacent to already colored nodes can be colored.

Program 6.7 shows the initial implementation for Graph Coloring. The code has been slightly extended to optimize the permutation order based on a greedy coloring algorithm. It is well known that the chromatic number can be determined exactly if the best possible order of nodes for this algorithm has been found. Employing Program 6.6 we compute the maximum clique for initializing the coloring process. First, because the size of any clique is –of course– a natural lower bound on χ. Then, because it turns out that the maximum clique is a good point for starting the coloring process. The resulting clique is written into a file, which is included as a solution prefix in the Graph Coloring solver. We also adapted a *selective policy* based on maintaining the remaining degree of uncolored nodes, with a preference given to choosing the ones whose number of colored neighbors is maximal.

Last, but not least, Program 6.8 provides the code for computing hitting sets. The problem setting is a bipartite graph in which a minimal selection of nodes (V) in the one set that covers all nodes in the other one (SET) must be found.

Program 6.5: DPLL helpers.

```
public class DavisPutnamLL {
    HashMap<String,Boolean> literalMap = new HashMap<String,Boolean>();
    class Clause implements java.lang.Cloneable {
      ArrayList<String> literals;
      Clause() { this.literals = new ArrayList<String>(); }
      void addLiteral(String literal) { literals.add(literal); }
    }
    boolean False(ArrayList<Clause> Clauses) {
       ArrayList<String> singleLiterals = new ArrayList<String>();
       for (Clause c: Clauses) {
          if (c.literals.size() == 1)
             singleLiterals.add(c.literals.get(0));
       }
       for(String sl : singleLiterals) {
          String neg;
          if(sl.startsWith("-")) neg = sl.substring(1);
          else neg = "-"+sl;
          for (Clause c: Clauses)
             if (c.literals.size() == 1 && c.literals.get(0).equals(neg))
                return true;
       }
       return false;
    }
    String pickLiteral(ArrayList<Clause> Clauses) {
       for (Clause c: Clauses) return c.literals.get(0);
       return "";
    }
    boolean Empty(ArrayList<Clause> Clauses) {
       for (Clause c: Clauses)
          if (c.literals.size() == 0) return true;
       return false;
    }
    void cutClauses(String literal,ArrayList<Clause> Clauses) {
       String cutLiteral;
       if (literal.startsWith("-")) cutLiteral = literal.substring(1);
       else cutLiteral = "-"+literal;
       for(Clause c: Clauses)
          c.literals.remove(cutLiteral);
    }
    void removeClauses(String literal,ArrayList<Clause> Clauses) {
       ArrayList<Clause> clausesToRemove = new ArrayList<Clause>();
       for (Clause c: Clauses)
          for (String l: c.literals)
             if (l.equals(literal)) clausesToRemove.add(c);
       for (Clause c : clausesToRemove) Clauses.remove(c);
    }
    String searchSingleLiteral(ArrayList<Clause> Clauses) {
       String literalToRemove = "UNIT";
       for (Clause c: Clauses)
          if (c.literals.size() == 1) {
             literalToRemove = c.literals.get(0);
             if (literalToRemove.startsWith("-"))
                literalMap.put(literalToRemove.substring(1),false);
             else literalMap.put(literalToRemove,true);
             break;
           }
       return literalToRemove;
    }
}
```

6.6 Experiments

We took problems of the known DIMACS benchmarks. As competitors we choose UCT (averaging leaf scores), NRPA (HD-NRPA, recursion level 5), NMCS (for nested Monte-Carlo tree search, recursion level 5), and SAT, calling the solver *Lingeling*, while applying a binary search on value k. The SAT solving process had a 1 hour timeout per instance and UCT was stopped after $100,000$ rollouts.

Program 6.6: Framework code for Clique.

```
class Game {
  int length;
  int rollout[V], visited[V];
  Game() {
    for (int j=0;j<V;j++)
      visited[j] = 0;
    length = 0;
  }
  bool terminal() { return length == V; }
  double score () { return length; }
  void play(int m) {
    rollout[length++] = m;
    visited[m] = 1;
    for (int j = 0; j< V;j++)
      if (visited[j] == 0 && !adjacent[m][j])
        visited[j] = 1;
  }
  int legalMoves (int moves[]) {
    int successors = 0;
    for (int m = 0; m < V; m++)
    if (!visited[m])
      moves[successors++] = m;
    return successors;
  }
};
```

Program 6.7: Framework code for Graph Coloring.

```
class Game {
  int length, colors;
  int rollout[V], used[V];
  Game() { colors = length = 0; }
  bool terminal() { return length == V; }
  double score () { return colors; }
  void play(int m) {
    if (m==colors) colors++;
    rollout[length++] = m;
  }
  int legalMoves (int moves[]) {
    int successors = 0;
    for (int j=0;j<colors;j++)
      used[j] = 0;
    for (int j=0;j<length;j++)
      if (adjacent[length][j])
        used[rollout[j]] = 1;
    int j=0;
    for (int j = 0; j < colors; j++)
      if (!used[j])
        moves[successors++] = j;
    moves[successors++] = colors;
    return successors;
  }
};
```

6.6.1 Clique

For maximizing Clique we executed the code shown in Program 6.6. We ran the NRPA$_\omega$ and NMCS$_\omega$ solvers (the subscript refers to finding the maximum clique size, and the superscript refers to using an open list) on specific DIMACS clique finding instances, which in the number of nodes are larger that the Graph Coloring benchmarks. We could solve many instances at or close to the optimum ω. Table 6.1 shows that, given 10 minutes for finding the largest clique (with some exceptions) NMCS$_\omega$ turns out to be better than NRPA$_\omega$ in larger instances, but worse in smaller ones.

Program 6.8: Framework code for Hitting Set.

```
class Game {
  int length, size;
  int rollout[V], visited[V], chosen[V];
  Game() {
    for (int j=0;j<SET;j++)
      visited[j] = 0;
    for (int j=0;j<V;j++)
      chosen[j] = 0;
    length = 0;
  }
  bool terminal() { return size == SET; }
  double score() { return length; }
  void play(int m) {
    rollout[length++] = m;
    chosen[m] = 1;
    for (int j = 0; j< SET;j++)
      if (!visited[j] && adjacent[m][j]) {
        visited[j] = 1;
        size++;
      }
  }
  int legalMoves (int moves[]) {
    int successors = 0;
    for (int m = 0; m < V; m++)
      if (!chosen[m])
        moves[successors++] = m;
    return successors;
  }
};
```

6.6.2 Graph Coloring

The few results of UCT_χ (the subscript now refers to finding the chromatic number) in the Tables 6.2 and 6.3 show that NRPA_χ, NMCS_χ and SAT_χ are superior. According to additional experiments this observation remains generally true for single-player UCT variants, when choosing not to propagate the average, but the maximum (or minimum) score, or when applying a robust criterion. We conclude that, compared to NRPA_χ/NMCS_χ, more implementation effort would be needed for UCT_χ to perform well.

Prior to its own search, the Monte-Carlo solver for Graph Coloring calls one above Clique solver NMCS^+_ω to compute the lower bound and to initialize the rollout with the enforced coloring. Moreover, for the Clique part in Tables 6.2 and 6.3 NMCS^+_ω (given at most 100 seconds CPU time) is better than the SAT solver (SAT_ω, given 1 hour CPU time).

The CPU time bound for NRPA_χ/NMCS_χ was set to 100 seconds for clique finding and 30 minutes for coloring. The results were close to optimal in many cases. In Table 6.2 SAT_χ almost always determines the chromatic number. For the harder results with several exceptions, we may conclude that NMCS_χ performs slightly better than NRPA_χ and SAT_χ .

6.6.3 Independent and Hitting Sets

With repeated computations of Independent Sets (choosing one color at a time) some of the upper bounds for Graph Coloring in Tables 6.2 and 6.3 can be improved. For the flat_?_? problems (given at most 15 seconds per iteration) we got an optimal bound 20 for the first problem in the sequence, as well as 34, 34, 92, 93, and 93 for the following ones. For queens_*n*_*n*, we could solve the problem 9×9 optimally, and for latin_square we found 111 as a slightly improved upper bound.

Moreover, a Hitting Set solver can help repeatedly finding vertex-disjoint independent sets.

Instance	ω	NMCS$_\omega$	NMCS$^+_\omega$	NRPA$_\omega$	NRPA$^+_\omega$
C125.9	34	34	**34**	33	**34**
C250.9	44	42	42	42	**44**
C500.9	?	53	52	51	55
C1000.9	?	59	60	57	62
C2000.9	?	66	66	62	65
DSJC1000_5	15	14	14	14	14
DSJC500_5	13	**13**	**13**	12	**13**
C2000.5	16	15	**16**	14	15
C4000.5	18	17	17	–	–
MANN_a27	126	124	125	125	**126**
MANN_a45	345	342	342	337	340
MANN_a81	1100	1090	1093	1085	1087
brock200_2	12	**12**	**12**	**12**	**12**
brock200_4	17	15	15	15	16
brock400_2	29	24	24	23	24
brock400_4	33	24	24	24	24
brock800_2	24	19	19	18	20
brock800_4	26	20	20	19	21
gen200_p0.9_44	44	39	**44**	41	**44**
gen200_p0.9_55	55	42	50	51	**55**
gen400_p0.9_55	55	50	50	50	52
gen400_p0.9_65	65	49	49	47	**65**
gen400_p0.9_75	75	71	71	72	**75**
hamming10-4	40	37	39	36	**40**
hamming8-4	16	16	16	16	**16**
keller4	11	**11**	**11**	**11**	**11**
keller5	27	**27**	**27**	26	**27**
keller6	?	50	52	49	53
p_hat300-1	8	7	**8**	7	**8**
p_hat300-2	25	22	**25**	**25**	**25**
p_hat300-3	36	28	28	35	**36**
p_hat700-1	11	10	10	9	**11**
p_hat700-2	44	30	41	41	**44**
p_hat700-3	62	48	48	59	**62**
p_hat1500-1	12	11	11	10	11
p_hat1500-2	65	57	57	54	**65**
p_hat1500-3	94	77	77	82	**94**

Table 6.1: Clique results.

6.7 Summary

Monte-Carlo Tree Search is a general exploration strategy that —similarly to SAT solving— leads to concise solver prototypes not only for games but also for many combinatorial optimization problems, including NP-hard graph problems. It can be used as a heuristic method for analyzing hypertexts, and computer or social networks. In fact, the scalable graph algorithms we considered apply to many other areas of computer science. For example hitting sets can be used to minimize the inspection points in motion planning as illustrated in Chapter 23.

Considering the simplicity of the code, the sequentiality of the execution on one CPU core, and the limited runtime invested per problem, the results on standard benchmarks are promising. The generic search framework helps to perform policy-based benchmarking.

With no or little problem-specific knowledge in the encoding, the SAT solver also performed well, especially for some Graph Coloring problems. This reflects the large amount of research and implementation efforts in a top-level SAT solver like *Lingeling*.

NMCS/NRPA intensifies the search with increasing recursion depth. The nestedness and policy refreshments relate to exponential restarting strategies known to be effective in SAT. We see there is much more common ground between the two solving approaches to be explored.

On the first glance, the outcome of the comparison of (simple) NMCS with (highly engineered) NRPA surprises. The implementation of NMCS is much simpler than the one of NRPA, but it frequently turned out to be the better solver. One interpretation is the following: if the policy as the learning object enjoys too few updates,

Instance	χ	SAT_χ	UCT_χ	$NRPA_\chi$	$NMCS_\chi$	SAT_ω
anna	11	**11**	**11**	**11**	**11**	11–31
david	11	**11**	**11**	**11**	**11**	11–29
games120	9	**9**	**9**	**9**	**9**	9–14
homer	13			**13**	**13**	
huck	11	**11**	**11**	**11**	**11**	10–25
jean	10	**10**	**10**	**10**	**10**	21–23
fpsol2.i.1	65	8–66		**65**	**65**	–
fpsol2.i.2	30	13–**30**		**30**	**30**	–
fpsol2.i.3	30	11–**30**		**30**	**30**	–
inithx.i.1	54	9–54		**54**	**54**	–
inithx.i.2	31	10–**31**		31–32	**31**	–
inithx.i.3	31	9–31		**31**	**31**	–
le450_5a	5	**5**		5–10	5–8	5–43
le450_5b	5	**5**		5–10	5–8	5–43
le450_5c	5	**5**		5–8	5–8	–
le450_5d	5	**5**		5–8	5–7	–
miles250	8	**8**	**8**	**8**	**8**	8–17
miles500	20		**20**	**20**	**20**	20–39
miles750	31	12–**31**	31.6	**31**	**31**	31–65
miles1000	42	11–**42**	42.3	**42**	**42**	40–80
miles1500	73	10–**73**	**73**	**73**	**73**	60-102
mulsol.i.1	49	18–**49**		**49**	**49**	27–89
mulsol.i.2	31	26–**31**		**31**	**31**	22–88
mulsol.i.3	31	25–**31**		**31**	**31**	24–89
mulsol.i.4	31	27–**31**		**31**	**31**	17–89
myciel3	4	**4**	**4**	2–**4**	2–**4**	**2**
myciel4	5	**5**	**5**	2–**5**	2–**5**	**2**
myciel5	6	5–**6**	**6**	2–**6**	2–**6**	**2**
myciel6	7	5–**7**	**7**	2–**7**	2–**7**	**2**
myciel7	8	5–**8**		2–**8**	2–**8**	2
queens_5_5	5	**5**	**5**	**5**	**5**	**5**
queens_6_6	7	**7**	9	6–**7**	6–**7**	**6**
queens_7_7	7	**7**	9.4	**7**	**7**	7–25
queens_8_8	9	8–**9**	10.7	8–10	8–**9**	8–28
queens_8_12	12	**12**	13.6	**12**	**12**	12–33
queens_9_9	10	8–**10**	12	9–11	9–11	9–33
school1	?	9–**14**		14–15	14–16	–
school1_nsh	?	7–**14**		14–17	14–17	–
zeroin.i.1	49	15–**49**		**49**	**49**	26–92
zeroin.i.2	30	22–**30**		**30**	**30**	28–85
zeroin.i.3	30	21–**30**		**30**	**30**	24–85

Table 6.2: Graph Coloring results for easier instances.

e.g., by very costly and/or long rollouts, due to its faster per-node performance, NMCS can be much faster in its exploration.

The take-home message is that a quick implementation of an NP-hard optimization problem for Nested-Monte Carlo search is often as simple as deriving a SAT encoding, and, with only a few lines of self-containing code, its resulting performance can be better.

The optimization of orderings is inherent to many combinatorial problems. For Graph Coloring, depending on the ordering chosen, the solution quality of the greedy algorithm varies between not being bounded by a constant and optimal. Given that the number of permutations $n! \geq (n/2)^{n/2}$ rises quickly, learning good permutation patterns in policies is challenging. MCS is one means to find such *needle in the haystack*.

An open problem is to find necessary/sufficient criteria for the convergence of NMCS/NRPA. While as in most MCS algorithms based on rollouts, we have *probabilistic completeness* in the sense that an optimal solution can always be found by chance. However, through nesting and adapting policies the success likelihood can become arbitrarily small, so that for now we cannot say for certain that the expectation of finding the optimum is equal to 1.

Instance	χ	SAT_χ	UCT_χ	$NRPA_\chi$	$NMCS_\chi$	SAT_ω
DSJC125.1	?	**5**	7	4–6	4–6	**4**
DSJC125.5	?	10–20	21.9	10–21	10–19	10–76
DSJC125.9	?	12–48	50	34–49	34–47	31–118
DSJC250.1	?	6–9		4–10	4–10	4–39
DSJC250.5	?	8–36		12–36	12–35	–
DSJC250.9	?	8–88		43–87	43–84	–
DSJC500.1	?	6–15		5–17	5–16	–
DSJC500.5	?	8–64		12–65	12–63	–
DSJC500.9	?	8–172		52–161	52–156	–
DSJC1000.1	?	6–26		5–27	5–27	–
DSJC1000.5	?	–		14–116	14–114	–
DSJC1000.9	?	–		56–299	56–293	–
latin_square	?	90–121		90–138	90–132	–
le450_15a	15	9–**15**		15–17	15–16	–
le450_15b	15	9–**15**		15–17	15–16	–
le450_15c	15	9–23		15–25	15–24	–
le450_15d	15	9–23		15–25	15–24	–
le450_25a	25	9–**25**		**25**	**25**	–
le450_25b	25	9–**25**		**25**	**25**	–
le450_25c	25	9–27		25–30	25–29	–
le450_25d	25	9–27		25–30	25–29	–
flat300_20_0	20			11–40	11–39	–
flat300_26_0	26			11–40	11–39	–
flat300_28_0	28	9–40		11–31	11-31	–
flat1000_50_0	50	–		15–113	15–112	–
flat1000_60_0	60	–		15–114	15–113	–
flat1000_76_0	76	–		13–114	13–113	–
queens_10_10	?	10–12	13.5	10–13	10–12	10–36
queens_11_11	11	10–13	14.4	11–14	11–13	11–41
queens_12_12	?	12–14	15.9	12–15	12–15	12–44
queens_13_13	13	9–16		13–17	13–16	13–49
queens_14_14	?	10–17		14–18	14–17	–
r_1000.1c	?	–		80–111	80–120	–
r_1000.1	20	9–21		20–21	20–21	–
r_1000.5	234	–		234–246	234–271	–
r_250.1c	64	9–67		64–67	64–66	–
r_250.1	8	**8**		**8**	**8**	7–14
r_250.5	?	8–66		65–67	65–66	–
r_125.1c	46	12–**46**		**46**	**46**	31–88
r_125.1	5	**5**		**5**	**5**	**5**
r_125.5	36	13–**36**		38	**36**	–

Table 6.3: Graph Coloring results for harder instances.

6.8 Bibliographic Notes

AI search often addresses solving NP-hard optimization problems [221] like number partitioning [415] or rectangle packing [351]. Even classics like the (n^2-1) Puzzle and Blocksworld are NP-hard when minimizing the number of moves [528, 580]. An initial set of 21 NP-hard problems has been provided by Karp [382].

As described in the previous chapter, the algorithm UCT belongs to the wider class of Monte-Carlo search (MCS) algorithms [86]. Besides solving specific one and two player games, variants of UCT are also best in playing general games [255].

There are single-agent variants of UCT, like SP-UCT, extending the above formula by some additional term, e.g., by $\sqrt{\sigma^2+D/n_i}$, where σ^2 is the variance in the results, n_i the number of simulations, and D another constant [554]. There are also different backpropagation criteria, like *average* for progressing the expected score, *maximum* for choosing the node with the highest score, *robust* for choosing the node which maximizes the lower confidence interval [553, 137].

Since Cook's encoding of a non-deterministic Turing machine as a Boolean formula in 1971, we know that SAT is NP-complete. SAT solving is a discipline on its own, with annual conferences and competitions. Starting with DPLL (see Chapter 1) there are numerous refinements like watch literals, and clause learning. Restarting strategies and heavy-tail distributions in SAT have been studied by [298]. Aspvall, Plass and Tarjan showed that

2-SAT is polynomial. There are interesting theoretical observations on 3-SAT such as the phase transition the clause-to-variable ratio of ≈ 4.5 (for random instances), and a SAT solver with running time $O(1.33^n)$.

The DIMACS benchmarks are described by [468, 111]. First results on greedy graph coloring can be found in [448]. Graph coloring with hitting set and *Tabu Search* (and a bound on the residual graph) are some of the best approximate coloring algorithms for the DIMACS benchmark [659]. Other cost models for the Clique problem have been proposed by [592].

Chapter 7
Multimedia Data

In terms of big data, multimedia content in the form of digital audio and video is one of the fasted expanding sources. This is accelerated by the growing resolution on fixed and mobile devices and given that the infrastructure for filming is present on almost every smartphone in our pocket.

The production of digital video is demanding, but during the process different errors can be introduced. Hence, quality control (QC) is mandatory for multimedia companies.

In this chapter, we develop and evaluate several approaches to feature extraction on sets of time-based events. On the one hand, these sets of events are extracted from video files and, on the other hand, manually annotated. By using methods of supervised machine learning the two sets of events will be mapped onto each other. After that, per time slot and requested event type, a binary classification will be applied. Thus, aspects of data mining and media technology will be discussed and combined with the goal to reach a reasonable reduction of the input-set by projecting it on an output-set. This will yield a save of operator-time in an automated process environment for quality control of audio-visual files.

7.1 Introduction

To[1] meet the aspects of quality assurance in media industry, available analysis tools are integrated in the process flow to support decisions. One question is whether a video file meets all relevant requirements to be broadcast, or whether further processing steps are needed.

Analysis tools rate the audio-visual material at different levels in the process. However, the baseband analysis remains a major challenge because the results of the various tools are always coupled with uncertainty. In addition, the accuracy, the recall, and the precision in individual analysis tools are often insufficient. Furthermore, tools from the professional sector are black boxes and provide no insight into their internal workings. Another problem is the syntax of the analytical results because there is no standardized notation being accepted among all tool manufacturers.

These facts make it difficult to make the right decisions in the process flow. Also, the prediction quality of the events varies not only from tool to tool, but also between different analysis focuses. Therefore, it is necessary to interrupt the process flow and evaluate the results manually, after passing the file through an analysis tool. Based on these inspections, decisions for successive actions are made.

The demanding question is whether there is a way through which a manual intervention could be accelerated by intelligent filtering of the analytical results, and to what extent an added value from supposedly independent results from multiple analysis tools with respect to the same audio-visual file can be obtained. Machine learning

[1] This chapter is based on joint work with Fritz Jacob. It improves the work from [204].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_7

Figure 7.1: Production of digital media files: tape input and analysis setup.

by data mining the tools' analyses can provide a basis for improved decisions. However, as the frame-based data is a time series of many, even noisy, events, the choice of features is not immediate, so that novel discrete approaches for feature extraction are derived.

Defects caused by the capture device or any other means during the ingest must be recognized early. Otherwise, this may cause a loss of video material, since a video tape has a limited lifetime, is possibly destroyed, or has been deleted after the import, so that only the faulty digital variant is left at the end.

Fortunately, there are professional multimedia QC tools. Even outdated video hardware can be utilized as it supports real-time error logs during playback of *Digital Betacam*-tapes. All these are tuned to find artifacts in the audio or video streams and result in a set of events of anomalies found in the input files, often associated with confidence values. However, the number of results and their quality differ significantly, so that manual QC often remains a time-consuming task for the operator.

Although the Digital Betacam recorders equally report read disturbances on the audio and the video head, it is a magnetic tape format for video, so that the focus of this chapter is video analysis.

We start with a general description of the problem we look at. Then, we address the essentials of the feature extraction modules, and evaluate the effectiveness of these approaches in a series of experiments.

7.2 Problem Formulation

In the broadest sense, we aim at a mapping between two sets of temporally correlated events in order to detect patterns in one of the event sets. An *event* e is a quadruple $e = (start, end, type, confidence)$ with $start, end \in \mathbb{N}$, $type \in C$, where C describes a set of event types, and $confidence \in \mathbb{R} \cap [0,1]$. The set of all events is denoted by $\mathscr{E}$. Each event $e \in \mathscr{E}$ refers to a *domain* in which it is situated.

A *domain* $D \in \mathscr{D}$ is a pair $(length, E)$ containing its duration *length* and a time-dependent set of events $E \subseteq \mathscr{E}$. The set of events in a domain $D = (length, E)$ can be partitioned into sets of *automated events* $A \subseteq E$ and *manual events* $B \subseteq E$ with $E = A \cup B$ and $A \cap B = \emptyset$. Furthermore, for domains $D_1, \ldots, D_n$ with $D_i = (length_i, E_i)$, classes $\mathscr{A}$ and $\mathscr{B}$ are defined as $\mathscr{A} = A_1 \cup \ldots \cup A_n$ with $A_i \in E_i$ and, similarly, for $\mathscr{B}$ we have $\mathscr{B} = B_1 \cup \ldots \cup B_n$ with $B_i \in E_i$, $i \in \{1, \ldots, n\}$. If $D = (length, E)$ and $E = A \cup B$, we impose that for all $e = (start, end, type, confidence) \in E$ we have $1 \leq start < end \leq length$, and for all $e = (start, end, type, confidence) \in A$ and $e' = (start', end', type', confidence') \in B$ we have $type \neq type'$.

Events in one domain can refer to identical type. Function $types : E \to A$ maps events to their according types, function $events : E \times C \to E$ projects events to ones of the chosen type, and function $\text{eval}_{A,b} : \mathbb{N} \to \{0,1\}$ is a mapping indexed by a set of events $(A, b) \in 2^{\mathscr{A}} \times B$ used for binary classification.

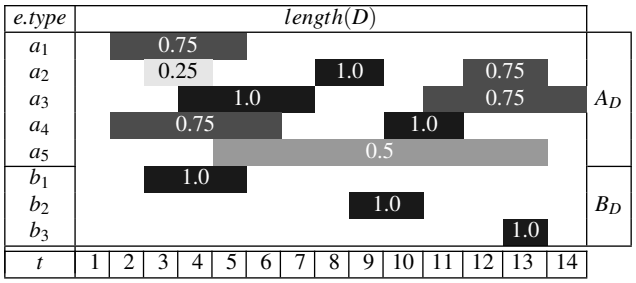


Figure 7.2: A sample domain with duration 14.

We are now ready to define the learning problem. Given a set of domains $\mathscr{D} = \{D_1, \ldots, D_n\}$ inducing sets of automated events $\mathscr{A} = \{A_1, \ldots, A_n\}$ and manual events $\mathscr{B} = \{B_1, \ldots, B_n\}$, the *event learning task* is to find a binary classifier $eval_{A,b}$ that with respect to set $A_i \in \mathscr{A}$ determines the existence of $b \in types(B_i)$, $B_i \in \mathscr{B}$ and $i \in \{1, \ldots, n\}$, for a given time step t.

Figure 7.2 depicts a sample domain D. In the upper part of the figure we see the temporally correlated occurrence of the events in A, and in the lower part the occurrence of the events from B. Each event contains its confidence parameter, which is also reflected in its gray scale (e.g., event $e = (2, 5, a_1, 0.75)$ can be found). In addition, we see that there are a total of eight different types of events in this image. Several events of the same type may co-exist in one domain (e.g., for a_2). The intersection of events of the same type, however, is prohibited.

For the events $e = (start, end, type, confidence) \in A$ we have $type \in \{a_1, \ldots, a_5\}$, and for the events $e' = (start', end', type', confidence') \in B$ we have $type' \in \{b_1, \ldots, b_3\}$. Moreover, $types(A) = \{a_1, \ldots, a_5\}$, $events(a_3, D) = \{(4, 7, a_3, 1), (11, 14, a_3, 0.75)\}$, and $eval_{A,b_1}$ is the following mapping from $\mathbb{N}$ to $\{0, 1\}$:

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$eval(A, b_1)$	0	0	1	1	1	0	0	0	0	0	0	0	0	0

An event $e = (start, end, type, confidence) \in A$ is *sliced* by event $e' = (start', end', type', confidence') \in B$ into an event $e'' = (start'', end'', type'', confidence'')$ if $start'' = \max\{start, start'\}$, and $end'' = \min\{end, end'\}$. Function $probe_{A,b} : B \to E$ is a *slice* of $A \subseteq E$ with respect to event $b \in B$. $\mathscr{P}$ is the set of all *probes* b. Moreover, function $pattern_{A,r} : B \to \mathscr{P}$ denotes a *pattern* to select a probe in set A with respect to event $b \in B$ and radius $r \in \mathbb{N}$. Figure 7.3 highlights $probe_{A,2}$ in dark and $pattern_{A,2}(9, 10.b_2, 1)$ in light gray.

An *interval* is an ordered pair (t_1, t_2) with $t_1 \leq t_2$ representing the set $\{t \in \mathbb{R} \mid t_1 \leq t \leq t_2\}$, and an *interval tree* I is a data structure supporting the operations $insert(I, n)$ to add an interval $n = (t_1, t_2)$ to I; $delete(I, n)$ to remove an interval $n = (t_1, t_2)$ from I; and $search(I, i)$ to locate an interval $n = (t_1, t_2)$ in I with $t_1 \leq i \leq t_2$.

For an event $e = (start, end, type, confidence) \in D$ we map t_1 to *start* and t_2 to *end*. An efficient implementation of an interval tree supports all the above dictionary operations efficiently in $O(\lg n)$ time, while requiring only $O(n)$ space. Note that when all intervals are to be reported in a stabbing query, the running time is output-sensitive and bounded by $O(k + \lg n)$, where k is the size of the output intervals.

7.3 Feature Extraction

The learning problem we aim at is a classification of multi-variate time series, where for all types c individual event sets are considered. To reduce the dimensionality of the learning vector, feature extraction is recommended. In the following, we discuss three different approaches, namely *fingerprints*, an *event correlation matrix*, and *naive statistics*.

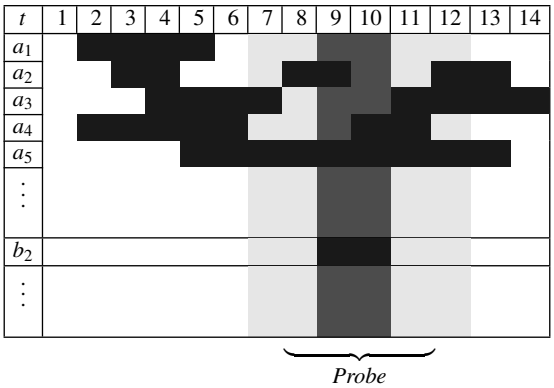
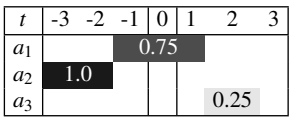


Figure 7.3: A *pattern* and a *probe* wrt event $(9,10,b_2,1)$, $A = \{a_1,a_2,a_3,a_4,a_5,\ldots\}$ and $r = 2$.

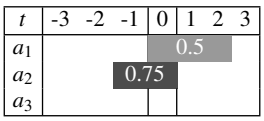
Fingerprints

A *fingerprint* is a data structure related to a probe, for which repeating combinations of events are generalized and used for classification. More formally, A *fingerprint* $f_{A,c,r}$: $\mathbb{N} \to \mathscr{E}$ with resepect to radius $r \in \mathbb{N}$ is a slicing of A with respect to a time interval, defined through an event type $c \in types(B)$, $B \in \mathscr{B}$. It consists of a *radius* r which defines the extension to both sides of its current median time point t. The set of all fingerprints is denoted by F. The function $radius : F \to \mathbb{N}$ can be used to access the extension of the fingerprint, while the *length* $|f|$ of the fingerprint $f \in F$ is defined as $1 + 2 \cdot radius(f)$.

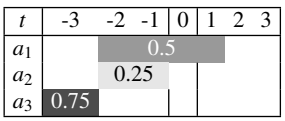
For matching a fingerprint f with set A we use an algorithm *performMatch* that loops on t (to the length of A) and k (to the radius of f). In the inner loop for each c in the slice with respect to k and t that is touched, the evaluation h is incremented by the weight w of k and t, times a certain combination of probability and confidence. The running time of this algorithm is $O(|A|^2 \cdot |f|)$.



(a) Base fingerprint 1



(b) Base fingerprint 2



(c) Base fingerprint 3

t	-3	-2	-1	0	1	2	3
a_1	0/3	1/3	2/3	3/3	3/3	1/3	0/3
a_2	1/3	2/3	2/3	1/3	0/3	0/3	0/3
a_3	1/3	0/3	0/3	0/3	0/3	1/3	0/3

(d) Intermediate result

Figure 7.4: Example of the superposition of fingerprints.

The *superposition* of the base fingerprints $f_1 \ldots, f_k$ with respect to events $a_1 \ldots, a_l$ operates in two phases. The relative occurence of the events is determined, followed by the multiplication with the confidence value average $confidence_f(a)$. For the example of Figure 7.4 we have $confidence_f(a_1) = 0.58$.

To increase the influence of values closer to the center, we apply Gaussian decay to both ends of the fingerprint. The *weight* of time step t and radius r is $weight(t,r) = e^{-4\cdot(t/r)^2}$.

For scoring we apply procedure *performMatching* to a set of manual events B (see Program 7.1). For each b in B, an individual score is computed, and the predicted values v are normalized to $[0,1] \subseteq \mathbb{R}$. Finally, we add the distance values between the predicted and the manual annotation $u \in \{0,1\}$, so that $score(u,v) = \sum_{1 \le t \le length} -|u_t - v_t|$.

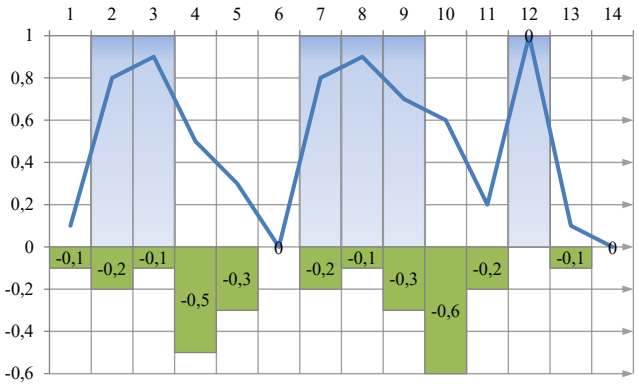


Figure 7.5: A *score* function for fingerprints.

Program 7.1: Perform matching of fingerprints.

```
procedure performMatching
Input: Event set A, fingerprint f
Output: Array V with prediction per time step

  for t = 1 to |A| do
    V[t] ← 0
    for each c ∈ slice_A(t) ∪ slice_f(center(f)) do
      V[t] ← V[t] + Prob_{f,t}(c) (∗ · confidence_f(c) ∗)
    for k = 1 to radius(f) do
      w ← weighting(k, radius(f))
      h ← 0
      for each c ∈ slice_A(t+k) ∪ slice_f(center(f)+k) do
        h ← h + Prob_{f,t+k}(c) (∗ · confidence_f(c) ∗)
      V[t] ← V[t] + h · w
      h ← 0
      for c ∈ slice_A(t−k) ∪ slice_f(center(f)−k) do
        h ← h + Prob_{f,t−k}(c) (∗ · confidence_f(c) ∗)
      V[t] ← V[t] + h · w
```

For the example plot in Figure 7.5 we have $score\ ([0.1,0.8,0.9,0.5,0.3,0,0.8,0.9,0.7,0.6,0.2,1,0.1,0],[0,1,1,0,0,0,1,1,1,0,0,1,0,0] = -2.7$.

Program 7.2 shows the steps necessary to generate fingerprints. The according evaluation function is encoded in Program 7.3. Program 7.4 shows the score function, which takes the result of *performMatching* as input.

Event Matrix

Another aspect for the generation of a model for feature extraction of a set of annotated video data is based on the following observation. Suppose an event of type $c \in types(\mathscr{B})$ frequently occurs at the same time as $c' \in types(\mathscr{A})$; we can infer that with high probability the interval of a c' event has to be annotated with c. Therefore, we use a event (correlation) matrix that denotes statistical relations between events from a given domain.

Let $k = |types(\mathscr{A}) \cup types(\mathscr{B})|$. The *event matrix* M of size k^2 is defined as $M = \{R_c \in \mathbb{R}^k \mid c \in types(\mathscr{A}) \cup types(\mathscr{B})\}$, where each R_c is a row in M and each column of R_c contains a pair of values (c', z) with $c' \in types(\mathscr{A}) \cup types(\mathscr{B})$, $c \neq c'$ and $z \in \mathbb{R}$. Assuming function $occur_{\mathscr{D}}(c) : C \to \mathbb{N}$ to count the number of events of a given type, and $matches_{D,q} : C \times C \to \mathbb{N}$ to return how often two types are present at the same time and q to be the threshold parameter for tolerance, we have $z = \sum_{D \in \mathscr{D}} matches_{D,q}(c,c')/occ_{\mathscr{D}}(c')$.

The *score* of an event matrix M with respect to event set A and $b \in types(\mathscr{A})$ is a function of time

Program 7.2: Computing fingerprints.

procedure createFingerprints
Input: **Domains** $\mathscr{D}$, **event types** c, **radius** r **and number** n **of combined fingerprints**
Output: **Fingerprints** $single_b$,$start_b$ **and** end_b

$P \leftarrow \emptyset$
for $D \in \mathscr{D}$ **do**
  for $e \in events_c(D)$ **do**
    append $\{pattern_{A,r}(e)\}$ **to** P
$Single \leftarrow Start \leftarrow End \leftarrow \emptyset$
for $p_e \in P$ **do**
  if $eventLength(p_e) = 1$ **then**
    append $\{convert(p_e, start(p_e))\}$ **to** $Single$
  else
    append $\{convert(p_e, start(p_e))\}$ **to** $Start$
    append $\{convert(p_e, end(p_e))\}$ **to** End
$single_b = evaluateFingerprints(Single, \mathscr{D}, c, n)$
$start_b = evaluateFingerprints(Start, \mathscr{D}, c, n)$
$end_b = evaluateFingerprints(End, \mathscr{D}, c, n)$

Program 7.3: *Evaluation* of fingerprints.

procedure evaluateFingerprints
Input: **Set of base fingerprints** F, **set of domains** $\mathscr{D}$, **event type** c, **number** n **of combined fingerprints**
Output: **fingerprint** *result*

$Q \leftarrow \{\}$
for $D \in \mathscr{D}$ **do**
  for $f \in F$ **do**
    $V \leftarrow performMatching(A, f)$
    $fingerprintScore \leftarrow calculateScore(V, events_c(D))$
    $Q[f] \leftarrow Q[f] + fingerprintScore \cdot |F|^{-1}$
sort Q **by values**
$resultList \leftarrow$ **select** n **leading entries of** Q
$result \leftarrow merge(resultList)$

Program 7.4: Computing the *Score* on basis of prediction values.

procedure calculateScore
Input: **Array** V **with predictions**, **set of von annotated events** E
Output: *score*

$U \leftarrow$ **array with length** $|V|$, **initialized with** 0.0
for $e \in E$ **do**
  for i = e.start to e.end do
    $U[i] = 1.0$
normalize P
$score \leftarrow score(V, U)$

$$score_{A,M,b}(t) = \prod_{e=(start,end,type,confidence) \in slice_A(t)} 1 + relOcc_M(b, type),$$

where $relOcc_M(c) : C \times C \rightarrow \mathbb{N}$ is the relative occurrence of accessing event c with respect to event c'.

Naive Statistics

We consider two more statistical features that are generated from the event data stream. Both rely on accumulating parameters for each time step t and can be used directly from a result set of A events. The *relevance* of a time step is the sum of confidence values, i.e.

$$relevance(t) = \sum_{e=(start,end,type,confidence) \in slice_A(t)} confidence,$$

where $slice_f : \mathbb{N} \to 2^C$ denotes the set of event types that are present at a certain time step. The *degree of parallelism* in an event set A at time step t is $parallel(t) = |slice_A(t)|$.

7.4 Evaluation

We first copied all video test data (with and without audio) to a Digital Betacam tape. Then, the tape was physically manipulated to enforce errors in the replay. Especially, we used diagonal bendings to generate *dropout* artifacts, see Figure 7.6. Horizonal bendings led to the loss of audio signal, and stronger manipulations to problems in the recorder, while perforation let to no significant change to the video.

Given that the video data was uncompressed, we converted it to IMX MPEG 50 and stored it into (so-called) MXF containers. The MXF data was forwarded to the different tools for analysis. Manual annotation has been done via a cutting program (see Figure 7.7). It features annotations via *markers*, linking a text to a frame or sequence of frames in an easy-to-handle manner.

First, we separated start from end fingerprints. For *constant color frames* we get the picture shown in Figure 7.8. For the learning process we use a set of annotated files and generate the following six different feature graphs for each of the files to be analyzed: *single-frame fingerprint*, *start-fingerprint*, *end-fingerprint*, *event matrix*, *accumulated relevance*, and *parallel events*. Every graph is a time series of discrete values for each frame. Based on the large amount of noise, we avoid thresholding, but instead take these as features to train a classifier (random forest). If g is the number of feature graphs and r is the radius of the frame to be classified we obtain $3g + (2r+1)g + 1$ attributes. For the six graphs above and a radius of 30 frames this results in 385 attributes.

As a feature of the algorithm, we learn different parameters for each of the five error types. Moreover, to control the experiment, we used 10-fold crossvalidation already on the file layer (to avoid inter-dependency of the features collected), and the standard statistical quality measures (accuracy, error rate, precision, recall, F1-measure). As the parameter space, we had 6 feature graphs (fingerprints, result correlation matrix, etc.), 4 analytical tools, n as the frame radius, together with $6k$ model-specific parameters for the fingerprints and l model-specific parameters for the result correlation matrix. With $n = 50$, $k = 20$ and $l = 10$ we have 56.7 million combinations, so that we performed the experiments in four stages: 1) for each of the five event types (macro blocking, digibeta drop outs, stripe blocking, edge blocking, and border blocking) 35 configurations were tested each (30 for fingerprints and five for event matrix correlation); 2) the frame radius for the best three configuration for each type is varied, yielding 75 further configurations; 3) additional feature graphs for the best three configurations are chosen yielding 15 further configurations; 4) for each event type we drop tools in their best setting, yielding 40 further configurations. For each of the steps a good parameterization is determined. Table 7.1 investigates the quality of the overall learning process.

If we relate these numbers to savings in operator time for 1m video, this is a significant advance. For manual QC we measured 30m working time and about 18 events. This is compared to the results of the four professional tools that in summary yield 671 events in about 15m. Checking one event might consume 3s, so that again 30m working time is needed. The learning scheme, however, resulted in only 22 events, which contained 66% of the manual annotated ones. This reduces the working time to about 1m. Note that the input set of the professional tools is also incomplete and contains errors and more than five types of events. While the training process consumes considerable time, classification is immediate.

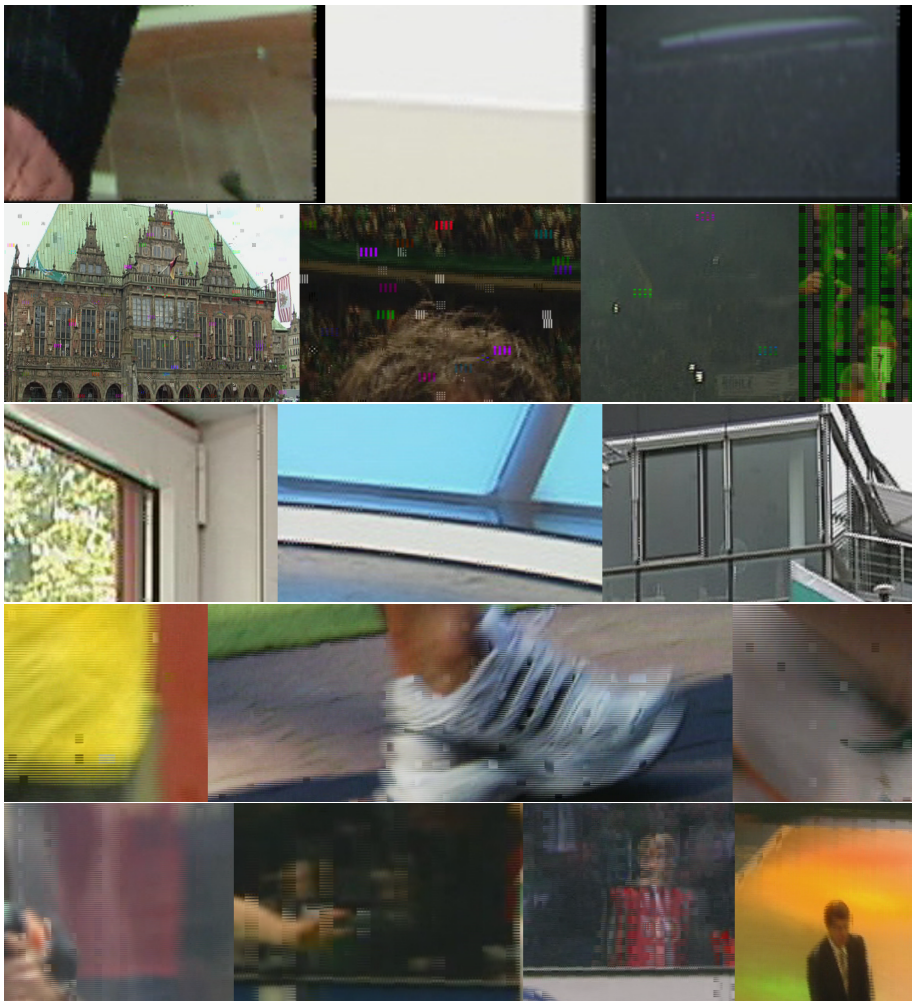


Figure 7.6: Artifacts, top to bottom: border blocking, dropouts, edge blocking, macro blocking, and stripe blocking.

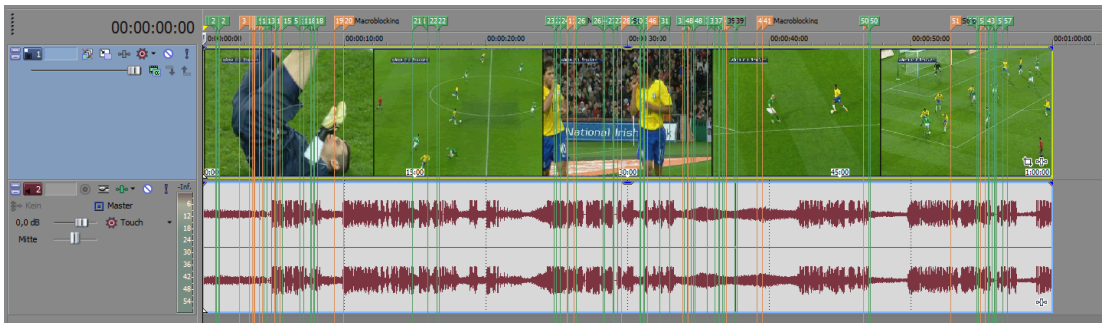


Figure 7.7: Manual annotation of video data.

7.5 Summary

We have looked at one problem of multimedia industry in general and broadcasting companies, which faces tremendous amounts of data: the semi-automated filtering and subsequent storage of digital video. Three ap-

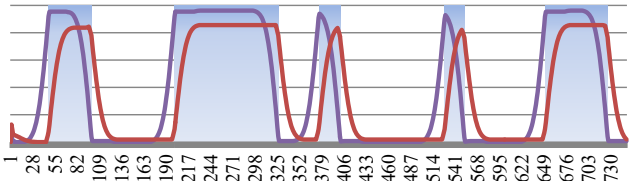


Figure 7.8: Effect of *start-end* fingerprints to detect constant color frames (start blue, end red, manual shaded).

Event Type	Number	Accuracy	F_1-Measure
Macro Blocking	199	0.79	0.58
Digibeta Drop Outs	135	0.91	0.76
Stripe Blocking	84	0.85	0.35
Edge Blocking	81	0.84	0.43
Border Blocking	41	0.81	0.3

Table 7.1: Results on learning.

proaches for feature extraction were developed and evaluated. The concept of fingerprints helped deriving typical automatic event combinations in certain manual event types. Furthermore, an event-matrix was introduced, through which the dependence on automatic to manual events can be mapped. Naive statistics aim at anomalities in the event data, regardless of the types of events. A combination of features for eventual classification was determined.

For the evaluation of approaches five manual event types were defined and applied to these various configuration parameters. The best result could be achieved with the event type *digibeta dropouts* (Accuracy: 0.91; F1: 0.76), the worst with *border blocking* artifacts (Accuracy: 0.81; F1: 0.3). This suggests that the selected event types are suitable for different classifications. Furthermore, for certain manual event types, individual analysis tools (e.g., Aurora results for *digibeta dropouts*) were sufficient, while for others a combination of analysis results (e.g., *Macro Blocking*) leads to better results.

We improved automated QC of audio-visual inputs by classifying event data of different analysis tools. In this way, an increase in the degree of automation could be obtained that is able to deal with uncertain and imprecise analysis results. How much we could reduce the amount of the automatic analysis results, depended on the (manually defined) event types. In principle, an arbitrary reduction is possible. The presented system can provide a significant relief of the employees, by reducing the event diversity and the fact that virtually no prior knowledge about the issues of analysis tools has been assumed.

The focus in dealing with the analysis results was on comparatively short events. For file-crossing events, both automatic and manual, there were far too few test data items. Only a part of the available amount of information of each event has been used since each event has different attributes, so that a direct comparison of several events of different kinds is no longer possible. In future, however, it would be useful to exploit this unused information, to find a way to make the process even more robust.

Apart from the used Digital Betacam tapes, learning data should also support other types of sources and formats used in the broadcasting industry. This requires the implementation of an appropriate infrastructure, as well as a simple user interface for annotation and review of video files and their analysis results ahead.

Rendering the on-line problem to deep learning is unsuitable, since there is no database, which can be used to train the network (the test data we collected is also insufficient). Additionally, we deal with heterogeneous features that when using the attribute-value pairs of event data are not readily available.

7.6 Bibliographic Notes

For companies in the media industry an increased level of automation is a key success factor [264, 405, 629]. One crucial operation is the ingest of digital or analog video tapes into a computer storage system for further post-processing and/or archival tasks [316, 144, 556, 103]. Supported by the *Bay Area Video Coalition*, the *A/V Artifact ATLAS* for the collection of errors in audio-visual material has expanded continuously and already contains many images and sounds for various types of errors. There is also a research department of the European Broadcasting Union, which —in cooperation with various broadcasting companies— published a list of QC recommendations which will be extended in the future. This set of QC items becomes accepted among broadcasting companies more and more.

There are two main terms for dealing with time-dependent data. On one side we have *sequential data*, as in [158]. This designation aims at the assessment of a given ordering on individual data, such as characters within a string. Intersections or the occurrence of parallel data cannot be represented. Also there is no time domain on which the individual data depends. These problems have already been mentioned by [14], who then proposed his formalism to describe time-dependent events. On the other site, we have the term *time series*, of which [272] provides a comprehensive overview of these, their areas of application and various methods for analysis. A large field of application of the time series analysis to predict based on trends, as in the stock market case [464].

When multiple, parallel time series are combined, we speak of *multivariate time series data*. To be able to predict failures of equipment in telecommunication networks, [652] developed a system for pattern recognition in time series of events, which is based on a genetic algorithm. The approach can only partially be transferred to the given problem, since this is a real-time application, and the predictions for the future to have to meet the already occurring events.

For feature extraction to segment audio files by [293] features are represented as time-series data for a binary classification task (segment boundary, no segment marker) with time windows of 100 ms. A way to extract useful features on the incoming event data to be then combined via a classifier has still to be found.

For data mining [484] found a method for the unification of time intervals to subsequently detect patterns in a database. Each entry contains a (larger) number of multivariate time series. The aim is to group entries based on the information contained in the time series. This approach can be applied to the given problem by casting several video files as a database, and by regarding the events of A as a time series. Although the approach identifies frequent patterns, certain events from B are not classified as such. Alternative approaches for the classification of certain patterns based on (multivariate) time series for various application areas other than the media industry have been proposed [37, 588, 282].

To automate finding and optimizing a suitable configuration to an event type as much as possible, adaptive boosting [557] can be applied to the extracted features. It also may be possible to use principal component analysis [174] for an additional reduction of the learning vector dimension. The ultimate goal is an unsupervised learning algorithm to detect common event combinations [482]. The event correlation matrix shares similarities with the covariance matrix [464, 258].

Chapter 8
Network Data

Probably the biggest amount of data these days is generated through the steadily increasing Internet traffic. Within this volume, incidents for network attacks have to be found and correlated.

We distinguish between information *safety* (protection against unwanted functionality) and *security* (protection against malicious actions). For this we need to know system and communication models as well as abilities of a potenial attacker, and have to determine the potential risks taken, while assuming a *state of compromise*; the higher the level of protection, the larger the according efforts.

There are basically two different detection methods distinguished in the area of Intrusion Detection Systems: misuse detection and anomaly detection. Detecting intrusion attempts by predefined patterns (also called signatures) is known as *misuse detection*. The different cases of possible misuses must be known to have appropriate patterns for the detection. E.g., misuses may be detected by code patterns in the payload of network packets or by violations against security policies. This detection method is known to be very reliable with respect to false positive alarms. However, the drawback is that incidents must be known in advance. This chapter studies a correlation method for Security Information and Event Management (SIEM) systems which is based on Conditional Random Fields and Tolerant Pattern Matching. This advanced machine learning method exploits expert knowledge to improve the detection accuracy of incidents, particularly addressing the problem of incomplete expert knowledge. It is shown that the method improves the detection accuracy of the market leading SIEM systems demonstrated by the correlation of events. Furthermore, it is shown that a Conditional Random Field is less sensitive to the imbalanced data problem than a naïve Bayes model. This will be helpful in using modeled knowledge in combination with learning from examples.

8.1 Introduction

Organizations[1] implement more and more business processes with IT systems and process sensitive data by applications. The threat against the IT infrastructure is a well-known and steadily growing problem. The attackers' motivations differ widely and range from stealing confidential data to gaining profit or harming an enterprise's reputation. Therefore, organizations need to protect their business-critical resources appropriately. One essential component of a comprehensive security management is to monitor the IT infrastructure at different levels, such as the operating system, network, and applications.

During the years, the complexity of IT infrastructures and, therefore, the amount of events to be processed have increased. Accordingly, the need for an efficient monitoring has grown. Additionally, enterprises are forced by governmental regularizations to ensure secured networks and data privacy, which is summarized under the term

[1] This chapter is based on joint work with Carsten Elfers, Hartmut Messerschmidt, and Otthein Herzog. It puts together and improves the work from [240, 239].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_8

IT compliance. Security Information and Event Management (SIEM) systems manage security-related events generated by different sources, like IDSs, firewalls, antivirus programs, and database logs, to correlate relations between them and to detect possible attacks.

Intrusion Detection and SIEM systems are often confused since they have the common objective to detect intrusions. However, SIEM systems go far beyond detecting intrusions. Technically, SIEM systems are used to gather events from different sources to generate an overview of the network status and to discern and report possible threats to a security operator. Event correlation is the major objective of a SIEM system for the purpose of detecting incidents.

There are many kinds of Intrusion Detection Systems which are related to SIEM systems by considering them as source or by using their methods in the SIEM correlation process. Mainly, they differ in their spatial coverage and their method of detection. Intrusion Detection Systems are distinguished with respect to their spatial coverage (i.e., the area under investigation) by Host Intrusion Detection Systems and Network Intrusion Detection Systems.

A host IDS is a monitoring software installed directly on a host, e.g., on a workstation or server. In contrast to network Intrusion Detection Systems, the HIDS is restricted to detect intrusions on the particular host it is installed on rather than detect intrusion attempts on all connected hosts. Considering only this aspect, the scope of detection seems to be limited. But with a direct access to the host, it is possible to use different and possibly more precise evidences — such as file integrity and CPU load information — than in the case of network-based intrusion detection.

Instead of analyzing a particular host (like an HIDS does), a network Intrusion Detection System (NIDS) analyzes network traffic on a network connection. This makes the maintenance easier than HIDS since the NIDS may cover a whole network and, therefore, it can be configured and deployed at a centralized point for the whole network under investigation. However, an NIDS does not not the insight into the hosts like an HIDS, which makes the detection of incidents more difficult or even impossible, e.g., on encrypted connections.

However, as attack data is sparse in the large amount of event data being processed by SIEM systems, sophisticated correlation approaches are needed. In this chapter, we introduce a correlation method based on Conditional Random Fields and Tolerant Pattern Matching. The system can generalize or abstract expert knowledge (by the use of Tolerant Pattern Matching) as well as to learn from examples given during the application of the system (by the use of the Conditional Random Field).

We compare the system with a leading SIEM system using events as input, and experimentally show that the presented approach has a smaller false-negative rate while producing a reasonable false-positive rate. In particular, the system can detect unknown attacks, which have not been modelled by a security expert, whereas a more traditional SIEM system needs exactly matching rules due to hard pattern matching. We also demonstrate that a Conditional Random Field is less sensitive to the imbalanced data problem compared to a naïve Bayes model, which is helpful in using modeled knowledge in combination with learning from examples.

Related methods have pros and cons: Anomaly detection methods have the advantage of detecting even unknown attacks which is of significant value to protect a network. However, anomaly detection typically suffers from an increased false positive rate of detection and the inability to explain the anomaly. Therefore, this kind of method does not fulfill the major requirements to incorporate expert knowledge for improving the detection accuracy. The missing interpretation of the anomaly delegates the task of explaining the anomaly to the security officer, which leads to an increased time to react that is avoided by the presented approach. Another problem inherent in the anomaly-based detection techniques is that an attacker might overcome the anomaly detection by performing only slow changes to a particular user profile, and that not all users can be expected to have a well-established behavior in the sense of anomaly-detection profiles. For example, some users may travel a lot and therefore have different working times on a local server. For this group of users, the anomaly thresholds for detecting an anomaly behavior is very high leading to a good entry point for possibly intruders operating below this high threshold.

In contrast, misuse detection approaches are unable to detect unknown attacks and are, therefore, dependent on a continuous update of their signature database, which violates the requirement of handling sparse and possibly

incomplete expert knowledge naturally apparent in expert systems. However, they have the advantages of giving interpretations of the observed incident by their signatures and of having typically a lower false positive rate.

Hybrid Intrusion Detection Systems are thematically close to enterprise SIEM systems since they are designed to process information from spatially distributed detection systems with differing detection methods. However, Intrusion Detection Systems are focused on detecting intrusions instead of discovering incidents as in the SIEM correlation process. In summary, some approaches meet partly the wanted requirements. However, no approach can be considered to perfectly match all.

8.2 Security Information and Event Management

Gaining a detailed impression of all SIEM solutions is out of the scope of this chapter. However, we want to introduce the market-leading product ArcSight.

The correlation engine of the ArcSight SIEM is ArcSight ESM, which uses an efficient pattern matcher to identify possible security risks by predefined rules. ArcSight uses the concept of filters and rules. Filters are sets of Boolean conditions filtering the event stream. Rules contain a left-hand side of Boolean conditions which may also contain filters. The right-hand side of a rule specifies the behavior in the case of a matching left-hand side, e.g., notifying an administrator by email. Event sources (e.g., event data from IDS sensors or firewalls) are connected via SmartConnectors to ArcSight. The events of the connectors are categorized (categorization is the same as semantics normalization) to be uniquely handled by the correlation process. Therefore, the events are categorized by Object, Behavior, Technique, Device Group, Device Type, Outcome and Significance. These categorizations are necessary for all incoming events to be correlated.

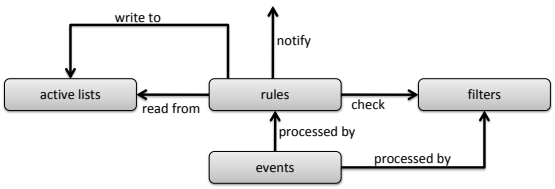


Figure 8.1: The ArcSight correlation process.

The ArcSight ESM Console offers several dashboard visualizations to help the security officer to investigate the network status, e.g., the system status, malware infections or log-in activities. ArcSight uses the concept of active lists to be stateful. Active lists are temporary lists used to store context information, such as informative correlations or suspicious events. Several active lists may be specified; each active list is filled by triggered rules which provide context information, e.g., for the evaluation of successive rules. Each active list has a predefined frame size to store the context information, e.g., for 24 hours. The correlation process of ArcSight is sketched in Figure 8.1. The normalized — or categorized — events are processed by rules and filters. The conditions of rules may contain filters as well as the content of active lists to evaluate their left-hand side. This is only a simplified view, rules may also refer to network models, asset models, priorizations, session lists, other rules or correlation data monitors. The rules' right-hand side can notify the user or write to active lists. The information in the active lists can be processed by further rules.

Most enterprise SIEM systems focus on rule matching (misuse detection) algorithms which suffer from similar problems as signature-based Intrusion Detection Systems: Previously unknown incident cannot be detected. However, some SIEM vendors, such as NitroSecurity, Unified SIEM, and QRadar, provide anomaly-detection to increase the detection accuracy with respect to new incidents. Nonetheless, the final decision making is mostly based on rules using anomaly evidence which still requires appropriate and comprehensive rule sets, but makes

these approaches more flexible with respect to detecting at least some varying incidents. Using anomaly detection in addition to rule-based detection reduces the problem of incomplete expert knowledge, but coincidently increases the number of false positives which contradicts the requirement of improving the detection accuracy. Furthermore, employing an anomaly detection method without the interpretation by rule-based correlation induces the problem of missing interpretations of the incident. Therefore, all these systems are focused on rule-based techniques and, in contrast to most Intrusion Detection Systems, all these systems are capable of using a variety of background knowledge, such as asset and vulnerability information. The major drawback of all these systems is that they require comprehensive and appropriate up-to-date rulesets to effectively detect incidents.

8.3 Anomaly Detection

An anomaly is an unexpected behavior of the network, maybe due to a malicious attack. System states are frequently measured and recorded in the form of time series. There are different sorts of anomalies, simpler ones are peaks in the time series, for which, usually, thresholds are applied. However, there might be periodicity in the data, due to daily work shifts, weekends, or public holidays, that has to be taken into account, too.

In automated time series analysis, there are many different known machine learning approaches. As a general distance metric on two time series, *dynamic time warp* applies, which is an extension of the *edit distance problem* and solved with dynamic programming.

Another idea is to discretize input time series data into a set of strings that is further processed. One option is *symbolic aggregate approximation* (SAX), illustrated in Figure 8.2: the discretization is achieved by imposing a series of breakpoints running parallel to the x-axis and labeling each region between the breakpoints with a discrete label.

Pattern matching then results in applying in form of substring matching. Known approaches, e.g., applied in Snort (and various virus scanners), invoke the multiple-pattern string matching algorithm of Aho and Corasick. The algorithm extends the known linear-time automata-based pattern matcher of Knuth, Morris and Pratt from one to a set of pattern strings. By the massive amount of raw Internet traffic data (PCAP), even for detection parallel versions of the approaches apply.

The two most promising anomaly detection approaches can be separated by their use of the data as either continuous or discrete quantities.

Exponential smoothing is used to predict future values for a time series. The next point in time can be singular, or, alternatively, several points in the future can be forecasted. Our interest is to predict the next value, which can be compared to the actual measurement of the sensor. This way we can decide whether or not an anomaly has been encountered.

An alternative proposal considers not only a fixed season but arbitrarily many. As a result, repeating patterns can be detected not only for weekly, but also for daily patterns. While in a weekly pattern every day stands on its own, this way a pattern can be found that, e.g., happens a) Monday-Thursday between noon and 6 pm, but b) on Fridays only between noon and 4 pm and c) not at all at week-end times. Those patterns, however, have to be learned by generated training data, and for each pattern we individually need a smoothing parameter that has to be determined.

Not all recorded time series contain regularities, so that it can be hard to finally judge about possible anomalies. To save resources in time and space, it may be recommended that some time series, when detected automatically, can be ignored. Even if the acquisition problem of the data itself can be cumbersome already, the main problem, however, is the impact of anomalies to the training data. Most approaches assume that the system data recording is initiated at a certain clock speed, which, unfortunately, is not always the case in our setting.

There are tools that solve this problem via interpolation, by means of the contraction of several data points. By the massive amount of data recorded, in the long run, the limited amount of memory becomes an issue. If

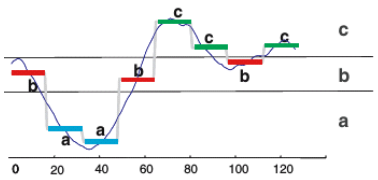


Figure 8.2: Transforming a time series into a string with Symbolic Aggregate approXimation (SAX).

there are repeating patterns in the data, compression techniques are applicable. One may solve this problem by storing only a fixed number of data points, so that the oldest data is replaced by the newest one. Such a database is called *round Robin database*. A less demanding problem is that in the test data there might be a time shift as patterns do slide in time.

More formally, a *time series* T of length n is an ordered sequence of pairs (t_i, x_i), $1 \leq i \leq n$, with $t_i \in \mathbb{R}_{\geq 0}$ being the *time stamp* and the $x_i \in \mathbb{R}$ being the *value measurement*. The *prediction* for T is a sequence x'_i, $1 \leq i \leq n$, with $x'_i \in \mathbb{R}$ being the *value prediction*. There are different predictions that act as *smoothed statistics* like the *simple moving average* $x'_i = (x_i + \ldots + x_{i-k+1})/k = x'_i + x_i/k - x_{i-k}/k$ or the *weighted moving average* $x'_i = (w_i x_i + \ldots + w_i x_{i-k+1})$ with $w_1 + \ldots + w_k = 1$.

For *smoothing factor* $\alpha \in (0,1) \cap \mathbb{R}$ the *exponential smoothing* $S_\alpha(T)$ according to Holt-Winters is a prediction defined as follows:

$$\begin{aligned} x'_0 &= x_0 \\ x'_i &= \alpha x_{i-1} + (1-\alpha) x'_{i-1}, \text{for } 1 \leq i \leq n. \end{aligned}$$

Values of α closer to 1 have less smoothing effect and give greater weight to recent changes in the data, while values of α closer to 0 have a greater smoothing effect and are less responsive to recent changes. To find an appropriate value of α the method of least squares can be used.

The moving average requires that the past k data points be kept, whereas exponential smoothing needs the most recent forecast value to be stored. Note that this incremental constant-time complexity will be dominated by other preprocessing steps.

For the time series analysis none of the above approaches has been selected, but an individual anomaly detection concept as a composition of different ideas. The basic assumption is that certain regularities are present in the data, so that deviations from those regular patterns can be detected even if they are still within the given tolerance bound. With respect to the given time step, they constitute an anomaly.

To keep the burden of configuration as small as possible, the temporal distance after which events are repeating is determined automatically. The more data we have, the more exact the prediction of the length of such pattern. Therefore, we expect an inaccurate prediction value for sparse data. Moreover, there is always the opportunity that in the data stream there is no regular pattern. If this is known a priori or found in the analysis a posteriori, such sensors are excluded from the time series analysis.

Based on a pattern length estimation, we can then predict in which interval the current measurement has to be located. Cumulating a certain tolerance to the interval bounds, an evaluation is returned that indicates whether or not an anomaly is present. If a repeated encounter of anomalies is found in a predefined time window, a message to the SIEM system is sent.

In order to predict the length of the pattern only on top of the measured data, the *power spectral density estimation* (PSDE) approach is extended. For this estimation the *periodogram* as well as the *auto correlation* of the time series are processed. Together they contain important information, but in singular are insufficient to predict the length of the pattern in a reliable way. Therefore, PSDE combines the two information sources and merges their individual strengths.

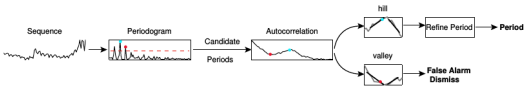


Figure 8.3: Steps in the PSDE algorithm.

A *periodogram* visualizes the intensity of frequencies and periodicity (pattern length) within a time series. For a fixed time series those values can be extracted by looking at the values on the x-axis. The advantage of the periodogram is that very simple values with large impact can be chosen. The disadvantage, however, is that for long periodicities (i.e., small x values) the predictions are rather inaccurate.

Auto correlation describes the correlation of a time series with itself, given that it is translated by value x. Therefore, the x-axis displays the length of the periodicity. The auto correlation counterbalances the weakness of the periodogram for long patterns.

Auto correlation has the problem that the most influential values cannot be easily extracted. Hence, in a first step, the algorithm uses the periodogram for selecting possible candidates. In a second step these candidates are checked and refined using auto correlation. These steps are visualized in Figure 8.3.

Both the trend and the outliers of a time series can have a huge impact on the approach. Therefore, in a preprocessing step the trend, if possible, is removed and the number of outliers is bounded. For computing the trend the *seasonal-trend decomposition procedure based on Loess* (STL) is called. As the pattern length has to be known in advanced, either the last known length is used, or the removal of the trend is disabled. In both cases, simple statistical outliers are removed by bounding them to 1.5 times the interquartile range.

In the experiments, the prediction of values using algorithms like ETS (error, trend, seasonal) or Holt-Winters were inappropriate since they adapt too rapidly to change and don't deviate substantially from the actual measurements (see Figure 8.4).

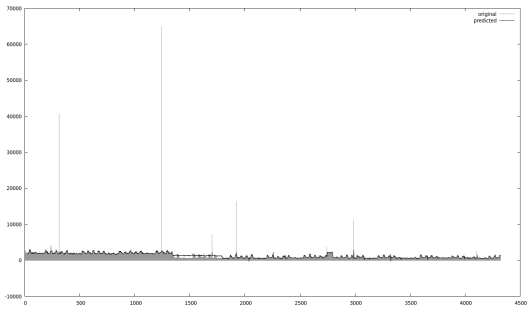


Figure 8.4: Holt-Winters (pattern length known, outlier removed).

Hence, for time series-based anomaly detection, TSAD for short, we go back to a relatively simple approach. The trend and repeating patterns are determined and subsequently added to predict the value of a current measurement. If the data is too sparse to guess the length of the pattern, the last value is used as the prediction (for the first value the same value is taken). In all other cases STL is used to compute a trend. Given that STL is robust against outliers, the data has not to be polished beforehand. An example of the STL-output is shown in Figure 8.5.

Next the trend is removed from the measurement data. The remaining values have to be polished by removing outliers. In this case, similarly to the PSDE, they are restricted to 1.5 times the interquartile range. Figure 8.6 shows the values of the different steps in one image.

The remaining data contains the pattern to be searched adjacently multiple times. To compress these values to a single pattern, all data points that match a data point in the pattern are averaged and selected as a representative (or canonical) element for the pattern. For the average, prefer the median to the mean to minimize the

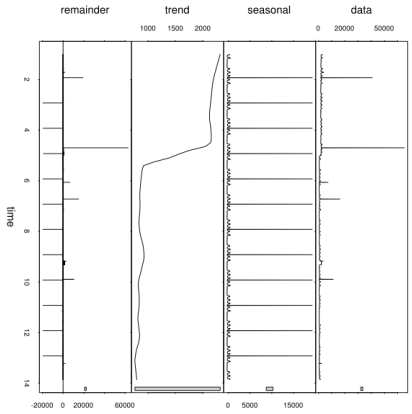


Figure 8.5: STL example.

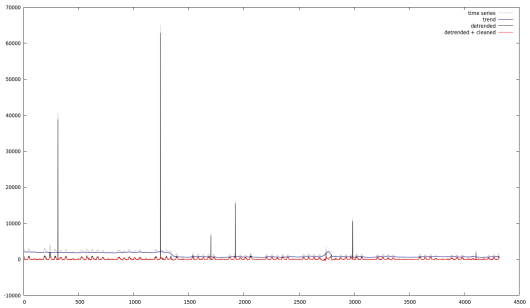


Figure 8.6: Time series with and without outliers.

effect of outliers. Given that the trend of the entire series of experimental measurements (including the current measurement) is known already, for this point the corresponding pattern value can be added to the trend. To cope with bounded noise in the data, a translation of two time steps, back or forth, is allowed. The reason is that regular duties that are not executed automatically, do not have to happen at the exact point in time. For example, a manual backup at the end of a working day, might happen regularly in between 5 pm and 6 pm, but with some tolerance in the exact execution. The value of the pattern (extracted from the five possible candidates) that together fit best the real measurement is taken as the prediction. The predictions of TSAD for the example time series are shown in Figure 8.7.

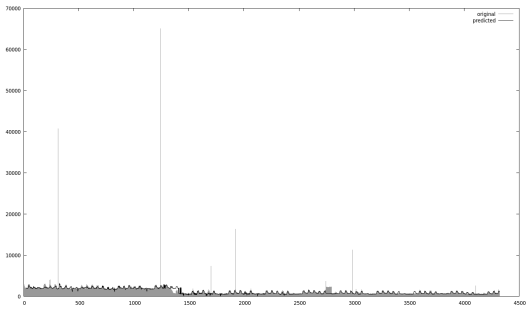


Figure 8.7: TSAD prediction (trend and averaged pattern).

Given that the predicted value rarely matches the exact measurement, it is recommended to define an interval in which a value can be found, without being labeled as an anomaly. Similar to the *aberrant behavior detection* (ABD) a confidence tube is chosen, that respects already recorded data. However, the error, how far the predicted value deviated from the exact measurement is again restricted to 1.5 times the interquartile range. If the value exceeds the interval, it is marked as an anomaly. As not all deviations correspond to a network attack, we go allow only a certain number of anomalies within a time frame. If this value is exceeded a warning message is returned.

8.4 Tolerant Pattern Matching with Background Knowledge

In the following, a hybrid approach for detecting incidents in the SIEM domain is introduced. Such approaches use different techniques to improve detection accuracy, such as preprocessing the data by clustering or by generating intermediate results for classification.

The approach of this work is split into a preprocessing part called Tolerant Pattern Matching and a post-processing part by a Conditional Random Field. The Tolerant Pattern Matching (TPM) approach presented here is a special kind of Soft Pattern Matching. The TPM term is used to clarify the difference from other Soft Pattern Matching approaches that do not use ontological representations, logical expressions, and generalizations of them in the matchmaking process. The post-processing part takes the matching values of the TPM as input for a statistical interpretation of incident hypotheses.

8.4.1 Preliminaries

Each observation $\boldsymbol{x}_j \in \boldsymbol{x}$ at position j in a sequence of observations $\boldsymbol{x}$ is assumed to be an event from an arbitrary source (e.g., from an Intrusion Detection System or a firewall) structured as (or transformed to) an attribute-value list, i.e., $(a,b) \in \boldsymbol{x}_j$ with the attribute a and the value b.

Further, an ontology is used to describe the domain characteristics. An excerpt from this ontology is given in Figure 8.8. The most general concept is called `Thing`, which generalizes all concepts such as `Analyzer`, `AggregatedAlertClassification` and `AlertClassification` (concepts are written in bold characters). More specific concepts of the `AggregatedAlertClassification` are `AdversaryActionClassification` and `AttackersKnowledgeGainClassification`. The individuals (concrete instantiations of concepts) of the ontology are written in italic non-bold characters. For example, `SCAN Amanda client-version request` is one individual which represents the original, raw classification text from a Snort sensor. All such raw classification texts are subsumed by the concept `AlertClassification`, associated to a sensor emitting this message by the relation `isGeneratedBy` (relations are visualized as dotted lines) and assigned to an individual aggregating (or categorizing, or normalizing) the concept by the relation `isAggregatedBy`.

The ontology is continuously growing by the input from several authors, e.g., by incorporating IT asset and vulnerability information. The core method of the correlation process, however, is very generic and capable of using a wide range of modeled background knowledge even beyond the currently modeled data.

8.4.2 Tolerant Pattern Matching

Tolerant Pattern Matching handles variations in the input data by generalizing patterns according to ontological background knowledge.

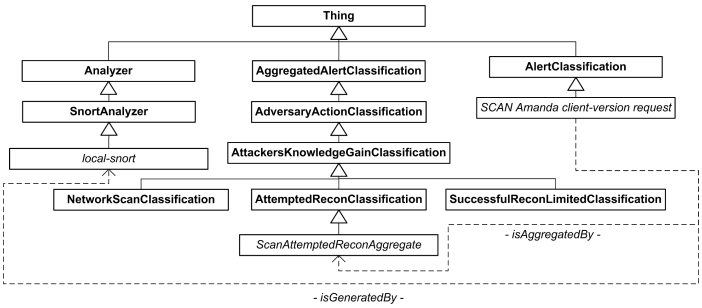


Figure 8.8: Excerpt of the ontology.

It is realized by successively generalizing the patterns and determining a residual degree of satisfaction with respect to the input data (the observations a.k.a. the events). A pattern consists of logical compositions of constraints. Each constraint is expressed by a relation between two entities.

Formally, an entity e from the set of entities $\boldsymbol{e}$ is either an individual, a concept, or a variable. A constraint $\gamma \in \boldsymbol{\gamma}$ is defined as $\gamma = eRe'$ of a left-hand side entity $e \in \boldsymbol{e}$, a right hand side entity $e' \in \boldsymbol{e}$ and a relation R from the set of all possible relations $\boldsymbol{R}$ between these entities. It is assumed that either e or e' is a variable or an individual fixed by an observation, i.e., e is replaced by b with $(a,b) \in \boldsymbol{x}_j$ if a refers to e (or e' respectively). A constraint γ is satisfied if there exists an interpretation $\mathscr{I}$ with $(eRe')^{\mathscr{I}}$.

A pattern p consists of a set of constraints and logical compositions among them. A partially matching pattern p – given the data $\boldsymbol{x}_j$ which is a set of attribute value pairs – is a real valued function with range $[0,1]$. The value of such a function is called degree of matching or matching degree.

$$p(\mathbf{x}) = \begin{cases} 1, & \text{if the pattern fully matches} \\ \alpha \in (0,1) & \text{if the pattern matches to degree } \alpha \\ 0, & \text{otherwise} \end{cases}$$

Each constraint in a pattern can be expressed as a query triple in a Description Logics query language like SPARQL.Let us consider an example. Let $\gamma_1 \in \boldsymbol{\gamma}$ be a constraint expressing that attribute `event` must have the value `ping` (which is assumed to be an individual in the ontology) and $\gamma_2 \in \boldsymbol{\gamma}$ be a constraint saying that the attribute `event` must be `port-scan`. Furthermore, the pattern p is specified as $\gamma_1 \vee \gamma_2$. Considering a query "Is pattern p satisfied for `event = port-sweep`?" the pattern query is transformed to SPARQL as

```
{ns:port-sweep owl:sameAs ns:ping} UNION {ns:port-sweep owl:sameAs ns:port-scan}
```

which is obviously not matching since a port-sweep is not the same as a port-scan or ping.

However, the pattern in this example expresses some reconnaissance events and if no other pattern is matching, this may give us a good hint of the kind of data presented to the pattern matcher. This can be easily achieved by using subsumption, i.e., each ping, port-scan and port-sweep may be subsumed by a concept called *reconnaissance*. E.g., γ_1, γ_2 or both could be abstracted to the condition that the event must be a reconnaissance instead of being a port-scan or ping. Each of these abstractions of the pattern will be matching. However, the smallest abstraction is desired to maintain most of the semantics of the original pattern, i.e., either γ_1 or γ_2 should be abstracted but not both.

In the following, a measure $\theta(\gamma^j, \gamma^k)$ for constraints γ^j and γ^k is assumed to quantify the similarity of an abstracted constraint γ from the original level j to an abstract level k. A simple example of such a measure is $\theta(\gamma^j, \gamma^k) = 1/(|j-k|+1)$. $\gamma^\perp$ denotes the original constraint on the most specific level $\perp$ (where $\perp$ is a positive integer) and $\theta(\gamma^i)$ is the short form of $\theta(\gamma^i, \gamma^\perp)$. This measurement is assumed to be 1 if the constraint is not abstracted and decreases if the constraint is getting more abstract by always being greater than or equal to 0. It can be considered as a similarity function, which says how exactly γ^j describes γ^k, or how similar the two

constraints are. Further, the properties of a similarity function are assumed to hold for the computation of the degree of matching of a pattern:

A similarity measure θ is a real-valued mapping into $[0,1]$ defined by the following properties:

$$\begin{aligned}
&\forall \gamma^j, \gamma^k : \theta(\gamma^j, \gamma^k) \geq 0 && \text{(positive definiteness)}\\
&\forall \gamma^j, \gamma^k : \theta(\gamma^j, \gamma^k) = \theta(\gamma^k, \gamma^j) && \text{(symmetry)}\\
&\forall \gamma^j, \gamma^k : \theta(\gamma^j, \gamma^k) \leq \theta(\gamma^j, \gamma^j) && \text{(identity)}\\
&\forall j < k : \theta(\gamma^j, \gamma^{k+1}) < \theta(\gamma^j, \gamma^k) && \text{(monotonicity)}
\end{aligned}$$

The similarity function values of the constraints are combined to a matching degree of the whole pattern by applying some fusion operator $F(\theta_1, \ldots, \theta_n)$ similarly to fuzzy pattern matching. This is necessary to consider the semantics of the logical operators while abstracting the pattern.

Therefore, a fusion approach is suggested by using the tree of logical operators in each pattern. The fusion function $F(p)$ of a pattern p is recursively defined with respect to some similarity function θ of constraints γ composed by logical operators as:

$$\begin{aligned}
F(\gamma_1^j \wedge \gamma_2^j) &= \min(F(\gamma_1^j), F(\gamma_2^j))\\
F(\gamma_1^j \vee \gamma_2^j) &= \max(F(\gamma_1^j), F(\gamma_2^j))\\
F(\neg \gamma^j) &= \begin{cases} 1 - F(\gamma^j), & \text{for } i = \perp \\ \beta \cdot F(\gamma^j), & \text{otherwise} \end{cases}\\
F(\gamma^j) &= \theta(\gamma^j),
\end{aligned}$$

where $\beta \in [0,1]$ is a penalty factor to additionally penalize the abstraction of negations.

β is a design parameter which is dependent on the used similarity function and the depth of the ontology. Optionally, a small β may be chosen without multiplying it with the similarity function, i.e., the similarity function is omitted in the case of negation since it directly abstracts to a tautology. The interpretation of the negation changes for the case of abstraction to ensure that an increasing abstraction leads to a decreasing fusion function value. This different interpretation results from the circumstance that the negation of an abstraction is a specialization (the complement of an abstraction). Therefore, the fusion function must be inverted for the negational case during abstraction.

The reason to choose the min-max fusion (of conjunction and disjunction) in contrast to a probabilistic interpretation is to avoid that patterns with a huge number of disjunctions have stronger tendency to be interpreted as true and patterns with a huge number of conjunctions strongly tend to be false. This makes the approach more sensitive to the number of constraints, which is undesirable in most cases. Furthermore, the min-max fusion allows an easy and common definition of temporal constraints as we will see later.

The fusion function F is monotonic with respect to θ. It builds a partial order of patterns regarding the generality of their containing constraints. From this basis, it is necessary to find the best matching pattern with respect to some input data, i.e., the matching pattern with the biggest F. How to find this solution efficiently is out of the scope of this chapter.

8.4.3 Divide-and-Conquer

In this section a divide and conquer algorithm is introduced to efficiently search for the most specific satisfied patterns, which correspond to the Pareto front of the constraint abstractions. Each level of abstraction of a constraint is represented as one dimension of the search space. The search space $\mathbf{X} = \{0, \ldots, n-1\}^d$ is divided

into satisfied elements (satisfied constraint combinations) $\mathbf{X}^+ \subseteq \mathbf{X}$ and unsatisfied elements $\mathbf{X}^- \subseteq \mathbf{X}$ with $\mathbf{X}^+ \cap \mathbf{X}^- = \emptyset$.

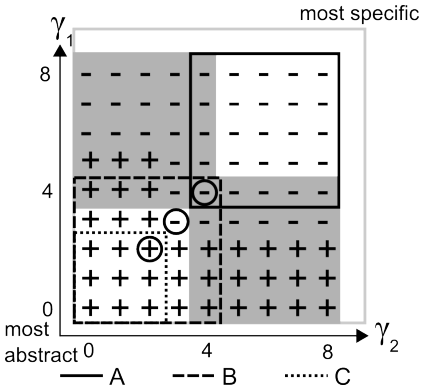


Figure 8.9: Example of the *Pareto* algorithm.

Figure 8.9 gives an example how the algorithm works for the 2D case (i.e., for γ_1 and γ_2). At first the middle of the search space is determined, i.e., point $(4,4)$. Around this point the search space is divided into (in the 2D case) four equal sized areas each including the middle and the corresponding border elements. Two of these areas are marked with a gray background the others as area A and area B. The minus sign at $(4,4)$ indicates that the pattern with γ_1^4 and γ_2^4 is unsatisfied, the circle indicates an inference call to test this satisfaction. Therefore, all more specific pattern combinations are omitted in the further recursion, i.e., area A. This method is continued for the gray areas but at first for area B. Area B is divided into four equal sized areas around the middle $(2,2)$. This is a satisfied match; therefore, we know that each more abstract pattern than γ_1^2, γ_2^2 is also satisfied, marked as area C which can be omitted in the following. The recursion is continued for the new middle $(3,3)$. At this point an unsatisfied area can be determined which also affects the currently not investigated gray areas. We know that from $(3,3)$ to $(8,8)$ every solution must be unsatisfied because they are more or equally specific. These temporary results are stored in a list and checked before investigating the gray areas in subsequent recursion steps to omit inference calls for these points.

The algorithm can be limited in the search space (by limiting the search depth) to give approximate results. By increasing the search depth the solution is more and more appropriately approximated.

Algorithm 8.1 shows the full implementation of the approach. This algorithm is initialized with an empty set of solutions (representing the most specific satisfied patterns) $\mathbf{S}^+$ and $\mathbf{S}^-$ (representing the most abstract unsatisfied patterns). The individual search spaces are specified by a most specific bound (**msb**) and a most abstract bound (**mab**), where **msb** and **mab** are coordinates of the search space. Initially, for all i we have $\mathbf{msb}_i = 0$ and $\mathbf{mab}_i = n$ (to ensure completeness **mab** is located outside of the actual search space). For reasons of simplicity, each constraint is assumed to have an equal amount of specializations/abstractions; however, the algorithm is also capable of differing amounts.

In Algorithm 8.1 we find *Eval*, the call to the reasoner. The other method that enumerates the sublattice structure is called *Hypercubenodes*(**msb**, **mab**) (without **msb**, **mab** themselves), formally defined as

$$\bigcup_{i=1}^{2^d-2} \mathbf{msb} \otimes bin(i) + \mathbf{mab} \otimes \overline{bin(i)},$$

where $bin(i)$ denotes the binary representation of a number i, $\overline{bin(i)}$ denotes its (first) complement, and $\otimes$ the component-wise multiplication of two vectors.

The following definitions express the previous considerations transferred to the d dimensional search space $\mathbf{X}$ which can be interpreted as a coordinate system. Let

Algorithm 8.1: Pareto algorithm for tolerant pattern matching

Input: Most specific bound $\mathbf{msb} \in \mathbf{X} = \{x_1, \cdots, x_d\}^d$
Most abstract bound $\mathbf{mab} \in \mathbf{X} = \{x_1, \cdots, x_d\}^d$
$\mathbf{m} \leftarrow \lfloor (\mathbf{mab} + \mathbf{msb})/2 \rfloor$
if $\exists \mathbf{x} \in \mathbf{S}^+$ **with** $(\mathbf{m} \geq_g \mathbf{x})$ **then**
  $s = 1$
else
  if $\exists \mathbf{x} \in \mathbf{S}^-$ **with** $(\mathbf{m} \leq_g \mathbf{x})$ **then**
    $s = 0$
  else
    $s = Eval(\mathbf{m})$
    if $s = 1$ **then**
      $\mathbf{S}^+ = \{\mathbf{x} \in \mathbf{S}^+ \cup \{\mathbf{m}\} \mid \forall \mathbf{x}' \in \mathbf{S}^+ \cup \{\mathbf{m}\} : \mathbf{x}' \geq_g \mathbf{x}\}$
    else
      $\mathbf{S}^- = \{\mathbf{x} \in \mathbf{S}^- \cup \{\mathbf{m}\} \mid \forall \mathbf{x}' \in \mathbf{S}^- \cup \{\mathbf{m}\} : \mathbf{x}' \leq_g \mathbf{x}\}$
if $\mathbf{mab} = \mathbf{m}$ **then**
  return
if $s = 1$ **then**
  $Pareto(\mathbf{msb}, \mathbf{m})$
else
  $Pareto(\mathbf{m}, \mathbf{mab})$
for each $\mathbf{h} \in Hypercubenodes(\mathbf{msb}, \mathbf{mab})$ **do**
  for $i = 1$ **to** d **do**
    $\mathbf{msb}'_i = \max\{\mathbf{h}_i, \mathbf{m}_i\}$
    $\mathbf{mab}'_i = \min\{\mathbf{h}_i, \mathbf{m}_i\}$
  $Pareto(\mathbf{msb}', \mathbf{mab}')$

$$\geq_g = \{(\mathbf{x}, \mathbf{x}') \in \mathbf{X}^2 \mid \forall i \left(\mathbf{x}_i \leq \mathbf{x}'_i\right)\}.$$

We say that $\mathbf{x} \in \mathbf{X}^-$ *dominates* $\mathbf{x}' \in \mathbf{X}$ if $\mathbf{x}' \leq_g \mathbf{x}$ and $\mathbf{x} \in \mathbf{X}^+$ *dominates* $\mathbf{x}' \in \mathbf{X}$ if $\mathbf{x}' \geq_g \mathbf{x}$.

All more specific patterns than an unsatisfied one are still unsatisfied and all more general patterns than a satisfied one are still satisfied. In other words, we have

$$\forall \mathbf{x} \in \mathbf{X}^-, \mathbf{x}' \in \mathbf{X}. \left(\mathbf{x}' \leq_g \mathbf{x}\right) \Rightarrow \mathbf{x}' \in \mathbf{X}^- \tag{8.1}$$

and

$$\forall \mathbf{x} \in \mathbf{X}^+, \mathbf{x}' \in \mathbf{X}. \left(\mathbf{x}' \geq_g \mathbf{x}\right) \Rightarrow \mathbf{x}' \in \mathbf{X}^+. \tag{8.2}$$

The algorithm computes the Pareto frontier, i.e., the set of extreme points $\mathbf{E} = \mathbf{E}^+ \cup \mathbf{E}^-$ with $\mathbf{E}^+ \cap \mathbf{E}^- = \emptyset$ containing each element of $\mathbf{X}^+$ with no element in $\mathbf{X}^+$ being more general,

$$\mathbf{E}^+ = \{\mathbf{x} \in \mathbf{X}^+ \mid \forall \mathbf{x}' \in \mathbf{X}^+ \left(\mathbf{x}' \geq_g \mathbf{x}\right)\}, \tag{8.3}$$

and each element of $\mathbf{X}^-$ with no element in $\mathbf{X}^-$ being more specific,

$$\mathbf{E}^- = \{\mathbf{x} \in \mathbf{X}^- \mid \forall \mathbf{x}' \in \mathbf{X}^- \left(\mathbf{x}' \leq_g \mathbf{x}\right)\}. \tag{8.4}$$

No element in $\mathbf{E}$ is dominated by another element in this set, i.e., the most compact representation of the set of satisfied / unsatisfied solutions.

The algorithm determines the whole set of satisfied constraints, i.e., $\mathbf{E}^+ = \mathbf{S}^+$.

Correctness To show the correctness of the algorithm we ensure that each element of the expected result set $\mathbf{E}^+$ is in the solution set $\mathbf{S}^+$ of the algorithm and, vice versa, i.e., $\mathbf{e}^+ \in \mathbf{E}^+ \Rightarrow \mathbf{e}^+ \in \mathbf{S}^+$ and $\mathbf{s}^+ \in \mathbf{S}^+ \Rightarrow \mathbf{s}^+ \in \mathbf{E}^+$.

We first show $(\mathbf{s}^+ \in \mathbf{S}^+ \Rightarrow \mathbf{s}^+ \in \mathbf{E}^+)$. If the search is exhaustive (this is shown later) we have that $\mathbf{s}^+ \in \mathbf{S}^+ \Rightarrow \mathbf{s}^+ \in \mathbf{E}^+$, since it computes $\mathbf{S}^+$ as $\{\mathbf{x} \in \left(\mathbf{S}^+ \cup \{\mathbf{m}\}\right) \mid \forall \mathbf{x}' \in \left(\mathbf{S}^+ \cup \{\mathbf{m}\}\right) : \mathbf{x}' \geq_g \mathbf{x}\}$, which is the same as the expected result $\mathbf{E}^+$ with $\mathbf{S}^+ \cup \{\mathbf{m}\} \subseteq \mathbf{X}^+$.

To see that $(\mathbf{e}^+ \in \mathbf{E}^+ \Rightarrow \mathbf{e}^+ \in \mathbf{S}^+)$ we investigate four conditions under which an element is inserted into (and kept in) the solution set of the algorithm $\mathbf{S}^+$. These conditions, directly derived from the algorithm, are as follows;

1. Each element from $\mathbf{S}^+$ must be contained in $\mathbf{X}^+$ which is the same condition as in the definition of $\mathbf{E}^+$.
2. The following assumption must hold for $\mathbf{e}^+$ to be inserted into $\mathbf{S}^+$,

$$\neg\exists \mathbf{x}' \in \mathbf{S}^+ . \left(\mathbf{e}^+ \geq_g \mathbf{x}'\right).$$

   This condition is not fulfilled if an equivalent solution $\mathbf{e}^+$ is already in the set $\mathbf{S}^+$ or if $\mathbf{e}^+$ dominates another element from $\mathbf{S}^+$. In both cases $\mathbf{e}^+$ is not inserted into the result set $\mathbf{S}^+$.
3. The next statement is

$$\neg\exists \mathbf{x}' \in \mathbf{S}^- . \left(\mathbf{e}^+ \leq_g \mathbf{x}'\right). \tag{8.5}$$

   This condition is always fulfilled, since we consider the case that $\mathbf{e}^+ \in \mathbf{X}^+$ and from Eq. 8.2 we know that this implies that $\mathbf{x}' \in \mathbf{X}^+$ which cannot be the case since $\mathbf{x}' \in \mathbf{S}^- \subseteq \mathbf{X}^-$.
4. We do not drop solutions because for all $\mathbf{m} \in \mathbf{X}^+$ we have Equation 8.3.

Analogically, the proof can be made for $\mathbf{E}^-$.

Completeness The recursion is omitted for $\{\mathbf{x} \in \mathbf{X} \mid \mathbf{msb} \leq_g \mathbf{x} \leq_g \mathbf{m}\}$ if $\mathbf{m} \in \mathbf{X}^-$ and for $\{\mathbf{x} \in \mathbf{X} \mid \mathbf{m} \leq_g \mathbf{x} \leq_g \mathbf{mab}\}$ if $\mathbf{m} \in \mathbf{X}^+$. This, however, does not affect the set of solutions due to the definition of domination and the definition of E that there should not be any value in the result set that is dominated by another element. Note that $\mathbf{m}$ has already been checked by the algorithm at this point.

The remaining space under investigation is getting smaller in each recursion path until $\mathbf{m}$ is getting equal to **mab** (the termination criteria). This is only the case if each edge of the space under investigation is smaller than or equal. This can be derived from the first line of the algorithm. At some time in the recursion the space of possible solutions is divided into a set of spaces with edges of the length one or less by still covering the whole space of possible solutions as previously shown. Further, if any point of such a smallest area is a possible solution (these are the corners), this point is under investigation in another space due to the recursive call with overlapping borders except of the borders of the whole search space at the specific borders due to there is no **mab** of any area including these specific border elements, e.g., there is no **mab** for the one element area $(8,8)$ in the example from Figure 8.9. For this border case the algorithm is called with a lifted **msb** to ensure that the unlifted specific bound is included in some smallest (one element) area as **mab** visualized as a light gray border in Figure 8.9. Therefore, each element of the search space which is a possible solution is investigated as a **mab** in some recursive path.

After computing the pareto front, the fusion function F is used on the remaining set of candidates to identify the most specific matching pattern abstraction.

8.4.4 Complexity Considerations

The worst-case running time is dominated by the number of calls to the reasoner. So we distinguish between the number of recursive calls $T(n)$ and the number of inference calls $C(n)$ (for the sake of simplicity, we assume $n_1 = \ldots = n_d$ and $n = 2^k$). Of course, a trivial algorithm testing all possible elements in S induces $C(n) = T(n) = O(n^d)$. We will see that the algorithm *Pareto* is considerably faster.

With $\lg n$ we refer to the dual logarithm $\log_2 n$, while $\ln n$ refers to the natural logarithm $\log_e n$.

For the number of recursive calls, let us first consider the 2D case. The number of calls of the divide-and-conquer algorithm in a 0/1 $(n \times n)$ matrix is bounded by $T(k) = \sum_{i=0}^{k} 3^i = \left(3^{k+1} - 1\right)/2$. Assuming $n = 2^k$ we have

$$T(n) = ((3^{\lg n} - 1)/2 = (n^{\lg 3} - 1)/2 = O(n^{1.5849})$$

For larger dimensions d the complexities $O(n^{\lg(2^d-1)})$ increases. In the limit for large d the exponent to n converges to d.

For the number of inference calls, again, we first consider the 2D case. We observe that the structure of the recursion corresponds to finding a binary search to the SAT/UNSAT boundary. The recursion depth is bounded by $\lg n$. Therefore, the worst-case number of calls to the reasoner of the divide-and-conquer algorithm in a 0/1 $(n \times n)$ matrix is defined by $C(n) = 2C(n/2) + O(\lg n)$. The $O(\lg n)$ term is due to the binary search. In the worst case the boundary between SAT and UNSAT cells is in the middle, where one quarter of SAT and one quarter of UNSAT elements are omitted.

Using the Akra-Bazzi theorem (a generalization to the well-known master theorem), the above recursion can be shown to reduce to $C(n) = O(n)$ as follows. For $k = 0$ it states that for the recurrence equation $T(n) = g(n) + aT(n/b)$ with $a = b^p$ we have the following closed form:

$$T(n) = O\left(n^p \cdot \left(1 + \int_1^n g(u)/u^{p+1}\, du\right)\right).$$

Here, $g(n) = \lg n = \ln n / \ln 2$ and $a = b = 2$ so that $p = 1$ and

$$T(n) = O\left(n + n \cdot \int_1^n \ln(u)/u^2 du\right) = O\left(n + n \cdot [-\ln u/u]_1^n\right) = O(n).$$

For larger dimensions d the complexities $O(n^{\lg(2^d-2)})$ rise. In the limit for large d the exponent to n converges to d.

8.4.5 Conditional Random Fields with Tolerant Features

Next, the combination of the previously presented Tolerant Pattern Matching approach with a post-processing by Conditional Random Fields is shown.

8.4.5.1 Tolerant Pattern Matches as Feature Function Values

The matches of the Tolerant Pattern Matching process are used as input features (i.e., the sufficient statistics) for a Conditional Random Field (CRF). A feature is built for each combination of label and pattern, i.e., $\boldsymbol{f} = \{f_1, \ldots, f_n\}$ with $n = |\boldsymbol{y}||\boldsymbol{p}|$, and $|\boldsymbol{y}|$ being the number of labels and $|\boldsymbol{p}|$ the number of patterns. Each feature matches exactly on one label and returns – in the case that the associated label is queried – the result of the fusion function of the associated less abstracted but matching pattern.

The combination of Tolerant Pattern Matching and Conditional Random Fields requires that a higher degree of matching lead to an increased influence on the posterior probability of the CRF. The intuition is that better matching patterns should more strongly account for the final decision making.

Therefore, we know that better matching patterns (i.e., features with a higher value) strongly account for the final decision making. Now, we are looking for a similarity function for the TPM approach which guarantees

that the best matching pattern (or in this context of CRFs, the best matching feature) dominates all other less matching patterns during the inference. This avoids that a huge number of slightly matching patterns overwhelm a strong (or even perfectly) matching one.

Therefore, it must be guaranteed that the best matching pattern exceeds the sum of all other less matching feature functions on the posterior distribution. Let $f_j^k = 1$ be the perfectly matching feature and all other $f_l^m = \varepsilon$ with $j \neq l$ be the less matching features by degree ε.

From the equation of Conditional Random Fields

$$Pr(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\lambda}) = \frac{1}{Z(\boldsymbol{x})} \exp\left(\sum_j \lambda_j f_j(\boldsymbol{y},\boldsymbol{x})\right) \tag{8.6}$$

we can obtain that for positive weights (i.e., $\lambda_j, \lambda_l > 0$) the following equation must hold:

$$\frac{1}{Z} \exp\left(\lambda_j f_j^k\right) > \frac{1}{Z} \exp\left(\sum_{l \neq j} \lambda_l f_l^m\right) \text{ for } m < k \tag{8.7}$$

$m < k$ expresses that f_j^k is less abstracted than f_l^m. We assume that both features have the same number of possible abstractions to make both features comparable.

It can be shown that for all cases we obtain the bound $1 > (|\boldsymbol{f}| - 1)\varepsilon$ with $|\boldsymbol{f}|$ being the number of feature functions. All partially matching feature functions must be less than the threshold $1/|\boldsymbol{f}| - 1$ with respect to a fully matching feature function. This solution can be treated recursively to ensure that the best matching partially satisfied feature function has the most influence on the posterior probability. Therefore, the fusion function F of a partially matching pattern $p^{(\perp - k)}$ must be ensured to be less than $\left(\frac{1}{|\boldsymbol{f}|-1}\right)^k$. Since all feature functions not matching on the label of interest are zero and the set of feature functions $\boldsymbol{f}$ is the product of labels and patterns $\boldsymbol{y} \times \boldsymbol{p}$, the number of feature functions $|\boldsymbol{f}|$ can be replaced with the number of patterns $|\boldsymbol{p}|$ for each label. The minus 1 of the denominator is omitted to ensure that the similarity function is below the threshold.

The similarity function $\theta(\gamma^k) = (1/|\boldsymbol{p}|)^{\perp - k}$ is defined for the number of abstractions $\perp - k$ of a constraint γ^k with respect to the number of patterns $|\boldsymbol{p}|$ by:

It is obvious that the fusion function F never exceeds this threshold due to the min-max fusion.

This similarity function ensures that the best matching feature dominates the sum of all less matching features with equal (or less) significance and with a comparable abstraction lattice.

8.4.5.2 Two Layers of Conditional Random Fields

After specifying the input for the Conditional Random Field (i.e., the similarity function, the fusion function and the pattern matches as feature values), we focus on the output, i.e., the labels of the CRF or the inference target. Two disjunct Conditional Random Fields are used, one for detecting and assessing threats (called Detection Layer) and one for explaining them (called Explanation Layer).

The *Detection Layer* (which can also be understood as Threat Layer) has three labels representing three threat levels a) dangerous, b) suspicious and c) normal. All patterns are used as input which results in $3|\boldsymbol{p}|$ feature functions (and weights). The Detection Layer is used to detect incidents out of the stream of events by determining the threat level. Further, this layer is essential for prioritizing incidents as we will see.

The *Explanation Layer* has one label for each modeled incident and is used for already detected incidents to explain the arbitrary steps belonging to the incident. The Explanation Layer is only used in succession of a high prioritized incident and, therefore, does not affect the computational efficiency of the incident detection. If $|\boldsymbol{i}|$ is the number of modeled incidents, there are $|\boldsymbol{i}||\boldsymbol{p}|$ feature functions in this layer.

There are two major reasons for splitting the detection and the explanation layer. The first is obviously a smaller inference effort for detecting incidents since the threat layer normally has a lower number of labels (only three). The second is the inability to derive the threat level from the explanation layer since the probabilities of the labels are not independent of the overlapping features. Figure 8.10 shows how these two layers are integrated in the correlation process.

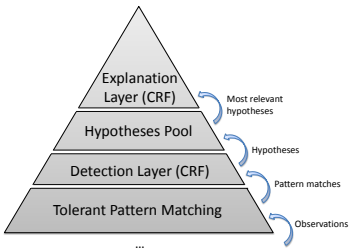


Figure 8.10: Visualization of the reduction of data from the bottom to the top during the correlation process.

8.4.5.3 Modeling Incidents

The prioritization of events is determined by their threats. Threats are not assigned to events (or pattern) directly since the threat may be related to the correlation of several events and to further background knowledge which is represented by pattern matches. The term *incident* is used to describe a set of pattern matches. The mapping $threat : \boldsymbol{i} \rightarrow \boldsymbol{t}$ is used to assign a threat level $t \in \boldsymbol{t} = \{N, S, D\}$ to an incident $i \in \boldsymbol{i}$. The following threat levels are distinguished: A *normal threat* (N) is assigned to an incident to indicate that the pattern matches triggering this incident does not indicate an increased threat. In other words, a normal threat indicates that the detected incident is a false positive and indeed is no serious incident. A *suspicious threat* (S) can be assigned if the triggering pattern matches are suspicious but do not indicate a high threat at this time. This threat level is used to mark potentially interesting situations which should be kept for further correlation. A *dangerous threat* (D) produces the highest prioritization and is used to indicate dangerous pattern matches.

Each pattern match is a pair of a pattern and a corresponding matching value. The following three matching values are possible: *match*: A match (M) indicates that the pattern must match for this incident; *mismatch*: A mismatch ($\neg M$) indicates that the pattern must not match for this incident; *unspecified*: Unspecified (U) describes that it is unknown if the pattern matches or mismatches and, therefore, does not affect the decision making.

From another perspective, each setting of pattern matches can be modeled to indicate one or multiple incidents. This is expressed by the mapping $incident : \{M, \neg M, U\}^{|\boldsymbol{p}|} \rightarrow \mathscr{P}(\boldsymbol{i})$.

Both mappings can be represented as a single two-dimensional matrix as visualized for three patterns p_1, p_2, $p_3 \in \boldsymbol{p}$ and two incidents $i_1, i_2 \in \boldsymbol{i}$ in Table 8.1.

Table 8.1: Example of an incident matrix relating pattern matches to incidents and threat levels.

	p_1	p_2	p_3	threat level
description	"port-scan"	"ping"	"source is admin"	
i_1	M	$\neg M$	$\neg M$	dangerous
i_2	$\neg M$	M	U	normal

As an example, assume that p_1 describes incoming port-scan events, p_2 ping events and p_3 that the source IP address belongs to a network administrator host. Incident i_1 is a reconnaissance incident and i_2 is a "normal" ping incident which typically occurs in the example network. For this setup, the matrix shows in the line of i_1 that a port-scan is dangerous if the administrator host is not the source of the port-scan. The pattern indicating a ping event is mismatching since two disjunct event types (i.e., ping and port-scan) cannot occur together. In the line of i_2 the matrix shows that ping events should be normal regardless of the source IP address.

8.4.5.4 Prioritization of Incidents

Let t be a certain threat level from $\boldsymbol{t}$ and j an index over the sequence of observations $\boldsymbol{x}$ with each observation (which is a set of pattern matches) $\boldsymbol{x}_j$. The probability of all observations belonging to a certain threat level is given by the inference of the Detection Layer Pr_{det} by:

$$Pr_{det}(t|\boldsymbol{x}) = \prod_{j=1}^{|\boldsymbol{x}|} Pr_{det}(t|\boldsymbol{x}_j) \tag{8.8}$$

Given a sequence of observations $\boldsymbol{x}$, the prioritization *prio* is determined by:

$$prio(\boldsymbol{x}) = \log_{10}\left(\frac{0.5Pr_{det}(D \cup S|\boldsymbol{x}) + 0.5Pr_{det}(D|\boldsymbol{x})}{Pr_{det}(N|\boldsymbol{x})}\right) \tag{8.9}$$

The prioritization compares the likelihood that all observations belong to an incident with dangerous or suspicious threat against the likelihood that all observations belong to an incident with a normal threat. This measurement is similar to the likelihood ratio as often used in sensor fusion approaches. Please note that a trade-off of false positives and true positives can be specified by a threshold for this prioritization.

This prioritization is used by the Hypotheses Pool. The objective of the correlation process is to determine a group of incidents which most likely are all dangerous. Each such group builds a hypothesis in the following, i.e., each hypothesis comprises of a sequence of events and incidents. Potentially, each permutation of the incidents may build a hypothesis. In practice, this is not feasible due to the exponentially increasing inference effort. Therefore, the Hypotheses Pool keeps the hypotheses with the highest priority and drops hypotheses with the lowest priorities to limit the number of hypotheses.

The introduction of Hypotheses allows one to define temporal relations in their scope. Besides the typical description logical constraints and logical compositions, patterns can describe temporal relations in their constraints in the scope of a hypothesis. This can be used to express dependencies over time, e.g., a failed login attempt may be considered more suspicious with a preceding port-scan. Therefore, the three temporal relations *currently*, *previously* $\ominus(\gamma(\boldsymbol{x}_j)) = \gamma(\boldsymbol{x}_{j-1})$ and *once* $\oplus(\gamma(\boldsymbol{x}_j)) = \gamma(\boldsymbol{x}_1) \vee \ldots \vee \gamma(\boldsymbol{x}_{j-1})$ can be used in the constraints of the patterns to express temporal relations.

The training of the Conditional Random Field is done by using Improved Iterative Scaling (IIS). The empirical probabilities for training are determined by the incident matrix. The matrix can be filled by modeled knowledge as well as by concrete examples of attacks which offers to use modeled expert knowledge as well as experienced misclassifications during the application of the system. This offers the ability to train the system during application, for example, to consider the individual network behavior of the application domain.

The modeling of incidents by experts often produces an artificial imbalance between benign and malign incidents. This occurs since incidents are modeled without the information about how frequent the incidents occur. This problem is reduced by learning from examples, but remains to be a challenge for the first deployment of the detection engine.

Another problem of CRFs trained with IIS is that they tend to overfit the data. In the original version of IIS, the model parameters are not limited and may even converge to infinitely huge numbers – which has obviously a

high impact on the posterior distribution. Therefore, regularization is typically used to overcome this problem, briefly, by tying the model parameters near to zero. In regularization, the model parameters themselves are considered as random variables with a specified prior distribution.

With these methods — IIS, regularization, and oversampling — we are well prepared to train the CRFs with examples as well as with modeled incidents.

8.5 Efficiency of Incident Detection

We evaluate the efficiency of the algorithm wrt the number of inference calls. In Figure 8.11 two tolerant matching algorithms and the result of a perfect guessing algorithm (a lower bound) are visualized. It is assumed that the lower bound algorithm checks exactly the Pareto border of satisfied and unsatisfied elements. Therefore, the best possible algorithm needs at least $|\mathbf{S}^+| + |\mathbf{S}^-|$ inference calls.

The divide-and-conquer algorithm *Pareto* with pruning the recursive calls as in Algorithm 8.1 – but without using the lists S^+ and S^- – is called **Pareto-0**. This is the first efficient algorithm one might think of. The algorithm is visualized as **Pareto** and the lower bound as **LOWERBOUND** in Figure 8.11 for the 2D case (no log-scale) and for the 4D case (log-scale).

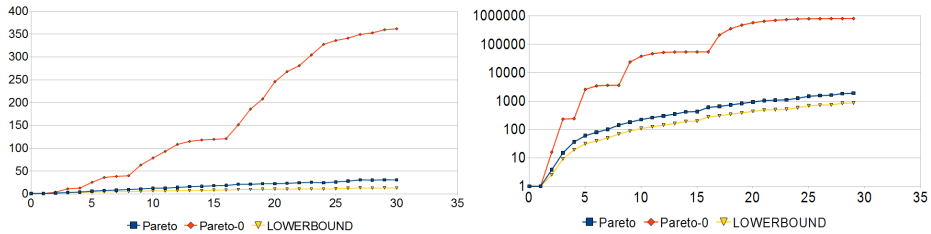


Figure 8.11: DL Reasoner calls in the 2D case (left), and in the 4D case (with logarithmic scaling). The x-axis represents the number of possible abstractions and the y-axis the amount of reasoner calls.

Both figures show that the inference calls of **Pareto** are located near to the optimal lower bound **LOWERBOUND** and considerably better than the typical divide-and-conquer algorithm **Pareto-0** in both 2D and 4D. These results are reasonable for the pattern matching algorithm due to the depth of an ontology being typically smaller than 30 and the patterns having typically a small number of constraints.

The amount of Pareto results, which is around the half of the **LOWERBOUND** value, is very small. For this set the degree of matching must be computed with respect to the fusion function to find the most optimal solution out of the set of Pareto-optimal solutions. This search can be done without to call the DL reasoner, since we already know that these solutions are satisfied.

8.5.1 Experiments with ArcSight

In the intrusion detection domain only a few benchmarking datasets are available and frequently used. A major reason for the low number of datasets is the confidentiality of real data including serious attacks.

The dataset A consists of 1,407 recorded malware samples. This collection of sample files is called dataset A containing just malign Snort events (predominantly `trojan-activity` events).

Benign traffic has been recorded in dataset B. One month (31 days) of traffic has been analyzed by a Snort IDS with the same rule set as used in the malware analysis (set A). The resulting dataset has been filtered by just using static IP addresses.

Three initial experiments have been made to test the detection technique of the market-leading ArcSight SIEM system against the approach of this work. Therefore, the datasets have been used to evaluate both approaches under laboratory conditions with most realistic data. The aim of this test was to evaluate the rule-based detection engine of ArcSight, and to evaluate if the approach of this work can improve the detection accuracy.

The ArcSight FlexConnector needs a mapping to map event captions to the appropriate ArcSight categorizations. Without these categorizations, ArcSight cannot detect any incident. Such a categorization has been developed for an actual Snort system. This mapping uses the classtype value of the Snort rules.

In the first test, two critical mappings of Snort `trojan-activity` events are different from the final version (after the lessons learned with ArcSight); these are Category Technique and Category Object. For testing ArcSight, the test data sets have been transmitted to the SmartConnector using this mapping. All built-in standard rules of ArcSight have been activated. The built-in `Security Activity` dashboard of ArcSight has been used to visualize the detection state. This dashboard shows detected attacks, Trojan-infected hosts, and worm propagations. With this first mapping, ArcSight was unable to detect any incidents in 43 randomly chosen test files. This poor performance directly led us to investigate the problem which could be identified in a rule that requires other categorization mappings.

In the second test, the mapping has been adapted by changing the mapping of the Trojan activity with respect to the category `Technique` and the category `Object`. Adjusting this led to a significant improvement in the detection accuracy. forty-two out of 43 malware samples from the dataset could be detected, i.e., a 97.7% true positive rate. Only one out of 31 noise samples from the dataset B has been detected as malware, i.e., a 3.2% false positive rate. This is a satisfying result, but it also shows the dependency on an appropriate mapping for the SmartConnectors. For a comparison, only the crucial rule has been (re-)modeled for the approach of this work. Since we additionally need the classification of benign events instead of just patterns describing malign events for this approach, the negation of this pattern has been assigned to a normal threat, too. The conclusion was that both detection engines came to the same results, ArcSight and the presented approach. Further, the test has shown that most of the malware could be detected by a simple rule expressing that all Snort events of the classtype `trojan-activity` are malware due to the dataset predominantly consisting of different `trojan-activity` events. This led to the third test with more demanding conditions on the correlation engines.

The event categorization mapping has been refined to distinguish between 1,048 categories instead of the previously 34 categories, i.e., ArcSight has been trained to distinguish different Trojan events. ArcSight has no built-in support for this fine granular distinction. However, we can add rules manually to support any category we desire. Since we know the samples containing malware and the samples without malware, we can create appropriate patterns to detect the events from these samples. In this test three different patterns (or rules) are used. Each pattern is a conjunction of conditions expressing that the category technique must have the appropriate value as specified in the malware sample files. Since the above approach additionally uses benign patterns, one frequently occurring event from the dataset B has been modeled to indicate harmless incidents. All other patterns are equivalent to the patterns of ArcSight (so both have the same starting conditions). Nine tests have been made with both approaches using the dataset to determine the true-positive rate. ArcSight has detected three of the nine attacks; in contrast, the illustrated approach has detected all nine attacks. To evaluate the false-positive rate, six samples from the dataset B have been used. None of the approaches has generated a false positive which shows that the approach of this work can outperform the detection method of ArcSight. However, these 15 tests cannot be understood as an empirical evaluation with statistically significant results. But the test shows that the approach at least yields a more flexible detection in some cases, whereas ArcSight relies on hard pattern matching which is only capable of detecting previously known incidents by perfectly matching rules and predefined thresholds.

8.5.2 Simulation Experiments

To generate statistically significant results, the ArcSight pattern matching method has been reimplemented to allow an automatic testing of both approaches. Further, a naïve Bayes and a Conditional Random Field approach compete. For the analysis of the detection performance, a probabilistic sampling method is used to generalize from a few samples to the whole population. Simple Random Sampling with replacement is used due to its acceptance for producing a representative evaluation and due to its simplicity. The test parameters are specified to guarantee a level of significance of 0.05 and a tolerable sampling error of 0.03, which means that the test result is in the interval of ± 0.03 with a probability of 0.95. It follows that at least 1068 samples are required to guarantee these test conditions.

We start with one benign sample (B) and one malign sample (A) for generating patterns. The generated patterns avoid using network specific characteristics like IPs and ports to ensure that the correlation uses characteristics from the different networks from which the samples are recorded. The test evaluates the approaches against one sample from the dataset which uses two hours (comprising typically less than 100) of malign events hidden in one day (with several thousands) of benign events. Specifically, each test result is collected by a) the modeling of benign incidents based on a sample of the dataset B, b) the modeling of malign incidents based on a sample from the dataset A, c) testing against benign events from the TZI dataset to measure true negatives and false positives, and d) testing against malign events hidden in the benign events from the dataset to measure true positives and false negatives.

In the Receiver Operating Characteristic (ROC) curve in Figure 8.12, seven test models have been compared using the same test and training data. A varying threshold over the prioritization function *prio* has been used to create the ROC curve based on 1,254 test results. The experiment uses the following seven models: 1) The naïve Bayes model $NB_{pm,os=1}$ using conventional (hard) pattern matching – indicated by the index *pm* – and an oversampling *os* which unifies the ratio of normal and dangerous incidents. The rate of normal to dangerous incidents is indicated by the real number next to the index *os*. 2) $NB_{tpm,os=1}$, the same model as in model number one, but with using Tolerant Pattern Matching instead of hard pattern matching which is indicated by *tpm*. 3) $NB_{tpm,os=1,\alpha=0.1}$, the same naïve Bayes model as number two, but with using Laplace Smoothing with $\alpha = 0.1$, i.e., adding 10% to the samples to smooth the distribution to avoid a so-called wipe-out in the naïve Bayes model which might occur by training with a low number of training samples. 4) The Conditional Random Field $CRF_{pm,os=1,\sigma=1}$ model using hard pattern matching, oversampling and a Gaussian Prior with $\sigma = 1$. 5) $CRF_{tpm,os=1,\sigma=1}$, as number four but using Tolerant Pattern Matching instead of hard pattern matching. 6) $CRF_{tpm,os=1,\sigma=5}$, the same CRF model as number five, but with a different parameter for the Gaussian Prior, i.e., $\sigma = 5$. 7) STRICT, a method only using conventional (hard) pattern matching without any probabilistic post-processing. This method can be assumed to perform equally than most enterprise SIEM systems.

In Figure 8.12 both probabilistic models, i.e., naïve Bayes as well as Conditional Random Fields benefit from using Tolerant Pattern Matching. Specifically, model five and six—using Conditional Random Fields and Tolerant Pattern Matching—significantly perform better than the STRICT method and the naïve Bayes models. In the area around the false positive rate 0.25, the gap between model five and six to the other models is quite large. This can be explained by the threshold used to generate the samples for the ROC curve. If the threshold deviates from zero, the influence of the Tolerant Pattern Matching is fading since partially matching patterns are producing less sharp probability distributions (to express their uncertainty) which results in prioritization values near to zero. This conjecture is underpinned by Table 8.2 showing the false positive rate for a threshold of zero. As we see, a threshold of zero leads to a false positive rate of 0.22 and a true positive rate of over 0.9. Model five has a 16% higher true positive detection rate than the STRICT method by keeping the false positive rate of 0.22. Further, we see that the CRF performs slightly better with a Gaussian Prior of $\sigma = 1.0$ than with $\sigma = 5.0$. Specifically, for lower false positive rates we see that using a smaller σ for the prior mostly leads to an improved true to false positive rate.

Tolerant Pattern Matching improves the detection accuracy for Conditional Random Fields as well as for naïve Bayes models. Conditional Random Fields with Tolerant Pattern Matching performs better than all the other tested methods. However, it is shown that a post-processing by naïve Bayes performs poorly with the given test

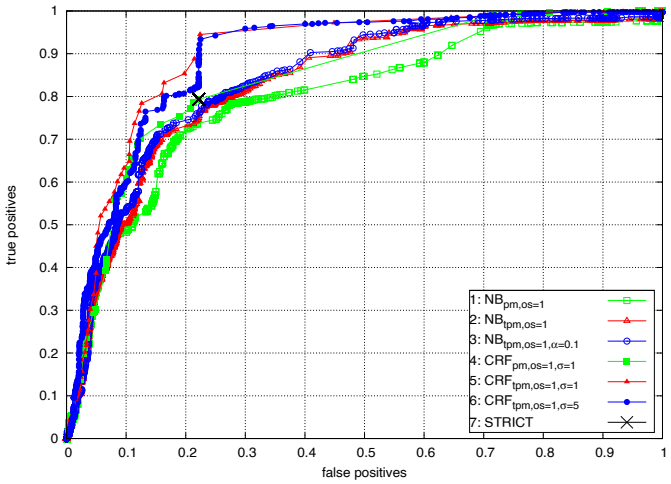


Figure 8.12: ROC curve with hard (models 1 and 4) and Tolerant Pattern Matching (models $2,3,5,6$). Models 1 till 3 use a naïve Bayes model and models 4 till 6 a CRF. The models are differently parameterized with respect to the Laplace smoothing α, the oversampling factor *os* and the regularization with a Gaussian prior σ. Model 7 is without any post-processing model.

Table 8.2: Results for the seven models given a threshold 0 of the prioritization function.

no.	model	tp	fp	precision	recall	accuracy	F1
1	$NB_{pm,os=1}$	0.82	0.42	0.66	0.82	0.7	0.73
2	$NB_{tpm,os=1}$	0.86	0.37	0.7	0.86	0.75	0.77
3	$NB_{tpm,os=1,\alpha=0.1}$	0.87	0.37	0.7	0.87	0.75	0.78
4	$CRF_{pm,os=1,\sigma=1}$	0.79	**0.22**	0.78	0.79	0.79	0.79
5	$CRF_{tpm,os=1,\sigma=1}$	**0.95**	**0.22**	**0.81**	**0.95**	**0.87**	**0.88**
6	$CRF_{tpm,os=1,\sigma=5}$	0.94	**0.22**	**0.81**	0.94	0.86	0.87
7	STRICT	0.79	**0.22**	0.78	0.79	0.79	0.79

parameters, even worse than using conventional pattern matching alone. This surprising result is investigated in the next test. Please note that the results are given for a SIEM level of detection, i.e., they are relative to the underlying events produced by the sensors, e.g., an IDS. In this case, based on the underlying Snort sensor we can obtain from Table 8.2 that the $CRF_{tpm,os=1,\sigma=1}$ correlation misses 5% of incidents that Snort has detected but also decreases the false-positive rate by 78%.

In the next experiment 1,077 tests have been evaluated. Each test consists of modeling two benign and two malign incidents. In contrast to the last tests, this one evaluates how the models behave when the trained modeled data are less sparse. Therefore, instead of one benign and one malign training sample two benign and two malign samples are used.

As expected, the detection rate of the *STRICT* method increases since more events are known to be malign. However, the false positive rate significantly increases as well as to obtain from Table 8.3. The approaches using TPM are not significantly varying with respect to the last test. This leads to the conclusion that their learning has already converged in the last test and, therefore, that they can more appropriately handle sparse reference data. In comparison with the *STRICT* method, the CRF/TPM approach can still more appropriately discriminate malign and benign incidents.

Table 8.3: Overview of the true positive and false positive rate of the *STRICT* method.

experiment	tp	fp	precision	recall	F measure
1	0.79	0.22	0.78	0.79	0.79
2	0.91	0.34	0.73	0.91	0.81

8.6 Summary

In this chapter we have seen a SIEM correlation technique by using Tolerant Pattern Matching and Conditional Random Fields. This approach challenges up-to-date enterprise SIEM systems by using hard pattern matching methods with respect to the detection accuracy. The rate of true positives has been improved by keeping the rate of false negatives. Further, we investigated that the discriminative Conditional Random Field approach is less sensitive to imbalanced data than a generative naïve Bayes approach which supports the synergy of learning from modeled expert knowledge and concrete examples. Additionally, the method addresses the problem of incomplete modeled expert knowledge (a.k.a. the knowledge acquisition bottleneck problem) and exploits ontological background knowledge to improve the detection accuracy, which has been shown to be a promising method.

8.7 Bibliographic Notes

Popular host IDSs are OSSEC, Tripwire, and Samhain. One of the most widely applied network IDS is the open-source software Snort [539]. This software performs fast signature matching on the packets of a network stream. Distributed Intrusion Detection Systems use both kinds of sensors to gain an increased spatial coverage without losing evidence. To give an example, the Distributed Intrusion Detection System (DIDS) [582] uses multiple instances of detectors monitoring hosts and LANs. Suspicious events from host and LAN detectors are sent to a centralized director which may request additional information from the detectors or process the events by a rule-based expert system. This method was extended for the application in wide area networks by [329] which is called Internetwork Security Monitor (ISM). Other well-known distributed detection systems are the Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [520] and the Graph-Based Intrusion Detection System (GrIDS) [589].

In 1998, a noteworthy intrusion detection framework [275] has been proposed by the CIDF working group called Common Intrusion Detection Framework [590]. In the latest release from the CIDF working group, they presented the architecture of an IDS with four modules for intrusion detection [521]. This distributed intrusion detection architecture is similar to the architecture of SIEM systems nowadays and shows the connection between (distributed) intrusion detection and SIEM systems.

An early example of misuse detection is NADIR [340]. NADIR is a rule-based expert system applied to the network of the Los Alamos National Laboratory. Later, two well-known misuse detection approaches were IDIOT [426] and STAT [519]. Dickerson et al. proposed to use fuzzy sets for misuse intrusion detection to be more robust with respect to variations in the data [157]. Ourston et al. [499] and Lee et al. [358] proposed to use Hidden Markov Models (HMMs) to detect multi-stage network intrusions.

Further, ontologies have been applied to misuse intrusion detection. Undercoffer et al. proposed an attack-centric ontology to be used by distributed Intrusion Detection Systems [626]. Based on this ontology they used rules (modeled in DAMLJessKB [410]) to specify intrusions and have shown by example how to detect distributed attacks. In [328], Chen et al. proposed a distributed system which uses a feature-centric ontology to formulate expert rules. The ontology is used to determine similarities between formulated rules and received sensor events to obtain a more flexible misuse detection.

Anomaly detection learns the normal behavior of a system — and its users — to report any significant deviations from this normal behavior. IT threats are common [606]. A single product, such as a firewall or an Intrusion Detection System (IDS) is not capable of fulfilling this security task [18, p. 665]. Therefore, large organizations, e.g., car manufacturers and financial institutes, have deployed Security Information and Event Management (SIEM) systems [494]. One example is the ArcSight Enterprise Security Manager (ESM) [22]. For pattern matching it uses a RETE [266] variant. Hybrid machine learning approaches have dominated the research of Intrusion Detection methods in the recent years [621].

One of the first anomaly detection methods based on audit trails was proposed in 1980 by James P. Anderson to support a security officer by monitoring exceptional behavior of computer users [16]. In 1994, Stephanie Forrest et al. presented innovative anomaly detection according to a human immune system [267]. for detecting foreign objects on a computer's hard disc. Neural Networks have been applied to anomaly detection, e.g., by Ryan et al. in their system called Neural Network Intrusion Detection (NNID) [549]. Before 2002, network anomaly detection has focused on header information of network packets. Kruegel et al. [425] first proposed to use the payload of network packets, specifically, the distribution of the payload, to detect anomalies in service connections for the detection of remote to local attacks.

Also, unsupervised methods, i.e., clustering techniques have been applied to anomaly intrusion detection. E.g., Eskin et al. [244, 672]. In 2004, Laskov et al. proposed a special kind of SVM for anomaly detection, called quarter-sphere SVM [439]. They investigated that quarter-sphere SVMs outperform plane and sphere SVMs in the KDD 1999 dataset. In 2007 Rieck et al. [537] presented anomaly detection by using n-grams of network traffic payload that was shown to outperform quartersphere SVM. Their model learns the normal behavior with prefix trees and detects anomalies using similarity functions. One advantage of this method is that it identifies the character sequences with the highest dissimilarity to normal character sequences. This might give an explanation of the attack, which is typically not possible in anomaly detection systems. Hidden Markov Models have also been applied to anomaly detection (e.g., [397]). The good performance of them has later been used to build a model by combining several Hidden Markov Models in the work of Khreich et al. [398]. For a comprehensive overview of further machine learning methods proposed in this context, please refer to [112, 275].

Hybrid Intrusion Detection Systems have been developed to get the best from both worlds, i.e., detecting known and unknown attacks with a low number of false positives and appropriate explanations. Early versions of hybrid Intrusion Detection Systems were IDES [462, 461] and MIDAS [567]. Specifically, in the IDES project Denning has shown how to use statistical models like standard deviation and mean to detect anomalies [154]. Nearly at the same time the Multics Intrusion Detections and Alerting System (MIDAS) [567] was developed to detect intrusions based on Denning's intrusion detection model [154]. IDES and MIDAS used the combination of anomaly and rule-based knowledge for the detection of intruders, which can be regarded as an early hybrid detection technique. However, MIDAS has a stronger focus on rule development as stated in [461].

There is a wide variety of distributed IDSs: Fan et al. [249] used a combined approach of detecting anomalies, misuses, and normal behavior. They trained a Decision Tree by RIPPER [132] with labeled attack and normal data. In contrast to other approaches, one classifier has been used to detect both anomalies and misuses. Lee and Stolfo [442] presented a framework called MADAM ID to automatically learn patterns and select relevant features from audit data. They developed anomaly detection for users and used a misuse detection model. In 2007, Hwang et al. [357] continued this work by using the generated misuse patterns of MADAM ID in the well-known Snort misuse detector. The combination of Snort misuse detection fed by an anomaly detection engine which mines patterns has also been proposed in [372].

Depren et al. [155] developed anomaly detection with Self Organizing Maps and a misuse detection with decision trees. Yu et al. [667] proposed to use Hidden Colored Petri-Nets (HCPN) to infer intrusions. Gupta et al. [314] proposed to use Conditional Random Fields (CRFs) for intrusion detection. This approach used normal data as well as abnormal data for training, which makes it a hybrid system. They showed that their CRF approach outperforms the usage of Decision Trees and a Naïve Bayes method.

The most frequently used benchmark is the KDD CUP'99 dataset [2] based on the raw tcpdump DARPA dataset. The KDD CUP'99 dataset consists of connection records and is well-suited for testing low-level Intrusion Detection Systems. However, this dataset is less appropriate for benchmarking SIEM systems on the higher

event level. The DARPA dataset is even on a finer granular level, i.e., raw tcpdump data, and, therefore, it is even less appropriate than the KDD CUP data for testing SIEM correlation engines. Further, several problems of both datasets have been investigated by Tavallaee et al. [610].

Imbalanced data is a serious problem for several machine learning approaches (see [38, 309]) and a known problem in the Intrusion Detection domain, too (see [114]). Batista et al. discovered that over-sampling methods are well-suited for imbalanced datasets [38]. Smith et al. discovered that regularization priors as Gaussian, Laplacian or Hyperbolic "perform roughly equally well" if they are appropriately parameterized [581]. Chen et al. agreed with that even while comparing further regularization techniques [116]. They also derived the gradient for Improved Iterative Scaling with a Gaussian prior. More details about parameter estimation with regularization can be found in [291, pp. 103].

Chapter 9
Image Data

We are addicted to the use of images, from personally taken shots, via magazines and scientific texts, to large-scale advertisements. Visualization is inevitable, *a picture tells more than a thousand words.* The sheer number of digital pictures, however, often exceeds the memory capacity especially on modern handheld devices, so that they are often to be stored on specific storage devices or in the cloud. Even video data is often reduced to the analysis of keyframes.

The automated process of making sense of pictures is referred to as *computer vision*. For vision applications arising in robotics and biometrics it is essential to classify what is present in the possibly segmented digital image. The challenges are manifold, including the size of the input, its shading, its orientation, or possible occlusions, just to name a few.

The power of fractal computation has been mainly exploited for image compression and halftoning. Here, we consider it for finding a fast approximate solution for the fundamental problem of nearest neighbor computation in the image plane. Traditional solutions use Delaunay triangulation for the case of optimal solutions or *kd*-trees for approximate ones. In contrast, we use a space-filling Hilbert curve which allows us to reduce the problem from 2D to 1D. It solves two nearest neighbor problems efficiently. We provide practical results on the accuracy of the method and show that it is significantly faster than a *kd*-tree.

Next, we present a simple but effective machine learning algorithm that we call *Bitvector Machine*: feature vectors are partitioned along the medians (in each component) and converted into bitvectors that are learned. It is shown that the method is efficient both in training and classification. The effectiveness of the method is analyzed theoretically for best and worst-case scenarios. Experiments on high-dimensional synthetic and real-world data show a performance boost compared to Support Vector Machines with RBF kernel. By tabulating kernel functions, computing medians in linear-time, and exploiting modern processor technology for advanced bitvector operations, we achieve a speed-up of 32 for classification and 48 for kernel evaluation compared to the popular library implementations. Especially for iso-oriented, multi-clustered problems the method has qualitative advantages over the linear classifier and achieves a high classification accuracy.

9.1 Introduction

Nearest[1] neighbor searches in the image plane are among the most frequent problems in a variety of computer vision and image processing tasks. They can be used to replace missing values in image filtering, to group close objects in image segmentation, or to access neighboring points of interest in feature extraction. In image filtering, the filter result is often computed only for a sparse set of key points. This is the case either if the

[1] This chapter is based on joint work with Martin Stommel, Thiemo Wiedemeyer, and Michael Beetz. It puts together and improves the work from [596, 223].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_9

processing of the whole image would take too much time, or if only a small set of pixels is suitable for processing (e.g., because of the *aperture problem*). The missing filter output must then be interpolated between the nearest keypoints. In image segmentation, nearest neighbor searches allow for a (possibly recursive) combination of close points to more complex objects, which leads to a fine-to-coarse decomposition of the image. For object recognition, the concept of spatial proximity is of fundamental importance, with a clear effect on the image statistics.

Traditional solutions to such nearest neighbor problems are a pixel-wise search within adaptive or fixed-size image windows, or the use of Delaunay triangulations and *kd*-trees. Search windows are unattractive because many irrelevant pixels must be visited. Fast results can only be achieved by using small windows or sub-sampling, often with a loss of accuracy. And although fast approximate results would often be preferred over accurate but slow computations, fixed window sizes may simply not suit the problem well. Balanced trees are more attractive because of their logarithmic run-time for a nearest neighbor search. The construction of the tree, however, introduces additional overhead, in the case of video sequences even repeatedly.

We will examine a fractal approach for a fast but approximate solution. The basic idea is to map the image plane to a one-dimensional space filling curve, the Hilbert curve, and perform the nearest neighbor search there. The Hilbert curve is known to keep the original 2D-relationship to a certain degree. As a result, an approximate nearest neighbor can be found by searching for the nearest neighbor (in other words the successor or predecessor) in a linear list. This can be done in one step or in log-log time depending on the implementation. Since the mapping is the same for every image (assuming a fixed size), there is no repeated overhead for video sequences. The theoretical and experimental results will show that the method is quite powerful in a computer vision context. Surprisingly, the use of space filling curves is largely unknown in this domain.

Due to the flexibility of kernel functions, statistical machine learning and pattern recognition with Support Vector Machine (SVM) are in frequent use. The training of SVMs is based on the maximization of a margin between the distributions of different classes. Noise tolerant loss functions allow negative margins for overlapping distributions. Common weights for outliers are linear (Hinge loss) or quadratic.

The dimensionality of the input data affects the speed of the classification as well, but it also often causes a numerical instability known as the *curse of dimensionality*. It is described as a general unreliability of distance computations for data sets where minimum and maximum distances approximate with rising dimensionality.

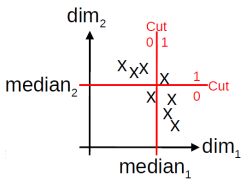


Figure 9.1: Effect of binarization.

In the application of machine learning algorithms to image and video data, a binary discretization of the popular SIFT (Scale Invariant Feature Transformation) has been applied. Although a feature binarization (see Figure 9.1) is a dramatic simplification of the input data, it has been observed for SIFT and SURF (Speeded Up Robust Features) features that the recognition accuracy (of SVMs and other learning methods) does not decrease significantly. The dimensionality of the feature vectors is 128 for SIFT (implemented as unsigned chars) and 64 for SURF (floating point values). The binarization compresses SIFT by factor 8 and SURF by factor 32 (given that in most SURF implementations 4 byte floating point data is stored). This problem is highly relevant to Robotics applications as SURF or SIFT are called at a high frequency in Simultaneous Localization and Mapping (SLAM).

We study the findings on feature binarization in a more general setting. A brief summary of Support Vector Machines allows us to introduce the method under the notion of a *Bitvector Machine* (the term *machine* links

to the fact that the classifier realizes a mathematical function, sometimes referred to as a machine). We discuss under which conditions the binarization of a feature vector is appropriate.

We observe that the Hamming distance is a natural distance estimate for Boolean vectors and has the property that its values are bounded by the length of the vectors to be compared. This allows a number of important code optimizations, most notably the use of look-up tables and native processor instructions for kernel evaluation. For transforming the input data from floating point to binary strings we exploit that medians can be computed efficiently.

9.2 Nearest Neighbor Computation

The method answers two types of nearest neighbor problems in three steps each: At first, the mapping between 2D and 1D-coordinates must be computed. For the all nearest neighbor problem, the set of keypoints S must be written as an array. The nearest neighbor assignment can then be done in two passes through the array. For the k-nearest neighbor problem, the set of keypoints S must be stored in a priority queue. The neighbors of a query point q can then be found by using the successor function of the queue. A Hilbert curve of recursive depth R subdivides the unit square into 2^R rows and columns. The size of the sub-squares is 2^{-R} x 2^{-R}. We map an image with unit square sized pixels (i.e., integer coordinates) to the unit cube by applying the scale factor 2^{-R}.

To make sure that an image with W columns and H rows fits into the unit cube, the recursive depth must be $R = \lceil \log_2 \max(W,H) \rceil$. We obtain C Hilbert indices range from 0 to $2^{2R}-1$. More recursions are redundant. With this scaling, we have a correspondence between the Hilbert indices of the 1D-curve, the sub-squares of the unit-cube, and the pixels of an image. Rectangular images and images whose side length is not a power of 2 do not cover the unit square completely. An image of size 640×480 would be mapped to a 1024×1024 grid in the unit cube. In this case, only 30% of the Hilbert indices correspond to pixels.

The result of the first step is a mapping $M(x,y) : N \times N \to N$ of image coordinates (x,y) to Hilbert indices, as well as the partial inverse mapping $M^{-1}(i) : N \to N \times N$ of a Hilbert coordinate to the image coordinate (x,y). The mapping must be computed only once for a fixed image size. For the whole image this can be done in linear time with respect to the number of pixels. We use arrays to store the mapping.

The mapping is used to solve two problems. In the *all nearest neighbor computation* we find the approximately closest keypoint $p \in S$ for all pixels $q \in I$. We will see that the *all approximately nearest neighbor problem* can be solved in precomputation time $O(C+n)$ and query time $O(1)$, while using $O(C)$ space.

Constant query time is achieved by computing a lookup table T of size $O(C)$ with the results. The indices of the lookup table are the Hilbert indices. At first, we mark the n keypoints S in the table, which is $O(n)$. In one forward pass, we compute for all indices i the distance $i-j$ to the nearest keypoint with lower Hilbert index j. To this end, we need to check if index i of the lookup table corresponds to a keypoint, i.e. if $M^{-1}(i) \in S$. In that case, we set the distance to zero. We also set it to zero if there is no preceding keypoint. Otherwise we increment the distance of the preceding table entry by 1. The check and the distance increment can be done in constant time, so the general complexity of the forward pass is $O(C)$. In a backward pass, we compute the distance to the nearest keypoint with higher Hilbert index. Then, we assign each index of the Hilbert curve to the nearest keypoint index from the forward and backward pass. The complexity of the backward pass is again $O(C)$. The total complexity is therefore $O(n+C)$.

The second problem is the *approximately k-nearest neighbor problem*. It can be solved in precomputation time $O(n \lg\lg C)$ and query time $O(\lg\lg n + k)$. The space requirement is $O(C \lg\lg C)$. We build a priority queue Q where all n keypoints S are ordered by their Hilbert index $M(p)$, $p \in S$. Using the precomputed arrays for M, the Hilbert index of a key point can be found in constant time. Inserting an element in a priority queue takes constant time plus an overhead of $O(\lg\lg C)$ for locating the right position in the queue using the successor function. The complexity for inserting n elements is therefore $O(n \lg\lg C)$. By linking all adjacent elements of the queue (in $O(n)$), we can find the successor and predecessor of an existing element in constant time.

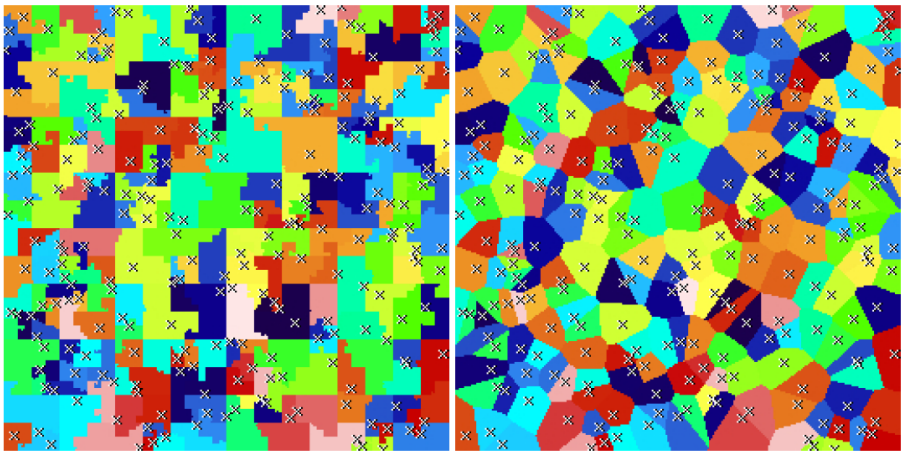


Figure 9.2: Nearest neighbor assignment using the approximative method (left image) and an exact search (right image) for 240 keypoints (marked by crosses). The resulting cells are colored randomly but consistent over both diagrams.

The priority queue needs to be computed only once for a given set of keypoints. The precomputation time is therefore $O(n \lg\lg C)$. For a query point q, the approximately nearest neighbor can be the successor of $M(q)$ in Q or its direct predecessor. The successor $M(p_s)$ can be found in $O(\lg\lg C)$. Its direct predecessor $M(p_p)$ is found in $O(1)$ using direct links. The comparison of the Hilbert indices ($|M(q) - M(ps)|$ and $|M(q) - M(p_p)|$) and the nomination of an approximately nearest neighbor is also constant time. The remaining $k-1$ neighbors can be found by reading out the directly linked $(k-1)/2$ predecessors and $(k-1)/2$ successors, which are single-step operations. The k-nearest neighbors can, therefore, be found in time $O(\lg\lg C + k)$ once the queue is constructed. The space requirement is $O(C \lg\lg C)$.

In order to measure the accuracy of the method, we solved the all nearest neighbor problem for a small image and varying numbers of keypoints first using the approximation and secondly using an exact method. It turns out that the approximation yields the same nearest neighbor as the exact method in about 50% of the queries (Figure 9.2). Even in the case where the method produces different results, the approximated neighbor is close.

9.3 Support Vector Machines

Raw data presented to a supervised statistical machine learning algorithm can be arbitrarily complex and is often mapped a set of numerical values, called the feature vector. The classification problem deals with the prediction of the labels l of previously unknown feature vectors $\mathbf{x} \in \mathbb{R}^d$ that constitute the test data. During training, a partitioning of the feature space $\mathbb{R}^d$ is learned, where each partition is assigned a label l from a small set L based on a set of training samples $(\mathbf{x}_1, l_1), \ldots, (\mathbf{x}_k, l_k) \in \mathbb{R}^d \times L$ with known label. The challenge is to approximate the unknown distribution without overfitting the training data.

Support Vector Machines achieve this task by learning coefficients for a kernel mapping to a high-dimensional space, where a linear class border is spanned by several support vectors that outline the data. We keep the presentation brief as there are textbooks on SVMs and related kernel methods. Theoretically, it should be sufficient to determine the class border by just three support vectors. However, it is not known in advance if any of the known kernels realizes a suitable mapping. The use of generic kernels instead leads to a much larger number of support vectors (which critically influence classification time). In the worst case finding a separating hyperplane takes quadratic time in the number of data points.

The classification rule for a two-class non-linear classification function ϕ is

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{s} \beta_i(\phi(\mathbf{x}_i)\cdot\phi(\mathbf{x})) + b\right), \tag{9.1}$$

where $\mathbf{x}_i$ is the ith of $s \le k$ support vectors, β_i is a coefficient that includes class label and Lagrange multiplier from the optimization, and b is some additional translation constant. Assuming a kernel function $K(\mathbf{u},\mathbf{v}) = \phi(\mathbf{u})\cdot\phi(\mathbf{v})$ we get

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{s} \beta_i \cdot K(\mathbf{x}_i,\mathbf{x}) + b\right), \tag{9.2}$$

which does not refer directly to ϕ, known as the kernel trick.

SVM training is a convex optimization problem which scales with the training set size rather than the feature space dimension. While this is usually considered to be a desired quality, in large-scale problems it may cause training to be impractical and classification to be time consuming.

The corpus of SVM applications is large and encompasses many areas of computer science. However, linear kernels $K(\mathbf{u},\mathbf{v}) = \mathbf{u}\mathbf{v} + g$ or Gaussian (RBF) kernels $K(\mathbf{u},\mathbf{v}) = \exp(-\gamma\|\mathbf{u}-\mathbf{v}\|^2)$, $\gamma \in \mathbb{R}$, are the ones that are most commonly used. In the second case, a Support Vector Machine is a function that itself defines a Mixed Gaussian Distribution and is related to Radial Basis Networks and Neuronal Nets with one hidden layer. Since the optimization objective is high accuracy instead of a low number of support vectors, a Mixed Gaussian input will be usually modeled by multiple centers per Gaussian in the SVM.

The running time for classifying one vector is $O(sde)$, where e is the time to evaluate the exponential. Practical SVM implementations might assume the input data to be normalized to avoid numerical difficulties.

9.3.1 Bitvector Machine

The approach we denote as *Bitvector Machine (BVM)* is a Support Vector Machine with a binarization (Boolean discretization) of the input: All vectors $\mathbf{x}_i \in \mathbb{R}^d$, $1 \le i \le k$, used in the training phase and all vectors evaluated in the test phase are mapped to $\{0,1\}^d$. The labels remain unchanged. The results are then fed into an SVM.

The *median* of n totally ordered elements is the element in the $\lfloor n/2 \rfloor$-th position after sorting. Linear-time $O(n)$ median selection algorithms are known. Let $\bar{\mathbf{x}} = (\bar{x}^1, \bar{x}^2, \ldots, \bar{x}^d)^\top \in \mathbb{R}^d$ be the component-wise median of the input vector, i.e. $\bar{x}^j$ is the median of the j-th vector components of $\mathbf{x}_1, \ldots, \mathbf{x}_k$. The BVM maps (training and test) vectors $\mathbf{z} \in \mathbb{R}^d$ to binary strings $\mathbf{x} = (x^1, x^2, \ldots, x^d)^\top \in \{0,1\}^d$ as follows: For each j, $1 \le j \le d$, we have $x^j = 0$ if and only if $z^j < \bar{x}^j$.

Computing all component-wise medians of vectors $\mathbf{x}_i \in \mathbb{R}^d$, $1 \le i \le k$, i.e. $\bar{\mathbf{x}}$, requires $O(kd)$ time. Thresholding all vectors $\mathbf{x}_i \in \mathbb{R}^d$ component-wise with $\bar{\mathbf{x}}$ can be executed in time $O(kd)$. Considering the input size of the data, the time $O(kd)$ is optimal.

To precompute the binarization for the training data we require median splits. If we were to split the data iteratively, we would build a kd-tree in $O(n \lg n)$ time, for which rectangular range queries take $O(\sqrt{n}+k)$ and membership queries take $O(\lg n)$ time. In contrast, in the Bitvector Machine the median splits are chosen independently of each other, one in each vector component. The binarization therefore defines a partitioning of the feature space into regions, where all separating hyperplanes intersect in one point. Geometrically, it can be interpreted as moving the origin of the Cartesian coordinate system to $\bar{\mathbf{x}}$ and representing each resulting orthant by a bitvector $\{0,1\}^d$ that indicates the position relative to the iso-oriented hyperplanes. The bitvectors correspond to nodes in a d dimensional hypercube.

Let ψ be the mapping that performs the binarization. We have

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{t} \delta_i \cdot K(\psi(\mathbf{x}_i), \psi(\mathbf{x})) + g\right). \tag{9.3}$$

Due to a different training, the number of support vectors t, weights δ and bias g might be different from the original values (s, β, b in Eq. 9.2). Moreover, as $K(\psi(\mathbf{u}), \psi(\mathbf{v})) = \phi(\psi(\mathbf{u})) \cdot \phi(\psi(\mathbf{v}))$, we see that we are actually dealing with a different kernel that transforms data via $\phi \circ \psi$ from $\mathbb{R}^d$ first into the Boolean space $\{0,1\}^d$ before lifting it into higher dimensions.

Because of the symmetry property, distance metrics based on an element by element comparison (like Euclidean or Hamming distance) in $\{0,1\}^d$ yield only $d+1$ different values. All possible results of the kernel $K(\psi(\mathbf{x}_i), \psi(\mathbf{x}))$ can, therefore, be precomputed.

For the Gaussian kernel $K(\psi(\mathbf{u}), \psi(\mathbf{v})) = \exp(-\gamma \|\psi(\mathbf{u}) - \psi(\mathbf{v})\|^2)$ the term $\|\psi(\mathbf{u}) - \psi(\mathbf{v})\|^2$ can only yield $d+1$ different values. For the Euclidean norm they range from 0 to d and equal the Hamming distance of the kernel arguments. By applying the parameter γ and the exponential to these values, the whole kernel can be precomputed and stored in a table. This avoids the repeated time-consuming computation of the exponential during classification. The Hamming distance can be computed by applying the population count instruction (counting the number of bits set) to the bitwise XOR disjunction of the arguments.

Precomputing kernels reduces training and classification time. Because training is done only once, we focus on the latter. Assuming d to be $O(w)$ for the computer word width w and native population count, the running time for classifying one bitvector is $O(t+d)$, where t is the number of support vectors. If population count is not native on the word level, then the classification of one vector hast the complexity $O(d + t \lg^* d)$, where $\lg^* d$ is the iterated logarithm, i.e. the height of the shortest tower of powers $2^{2^{\cdots}}$ that equals or exceeds d. This assumes a large word width w and is mainly of theoretical interest.

Computing the binarization $\psi(\mathbf{x})$ of the test vector $\mathbf{x}$ takes time $O(d)$. The population count and XOR to be executed on the word level to compute the Hamming Distance run in $O(1)$. Given that the kernel is tabulated, we require only lookups to the kernel table, so that multiplication with a constant and addition have to be executed t times to evaluate the classification formula $\sum_{i=1}^{t} \delta_i \cdot K(\psi(\mathbf{x}_i), \psi(\mathbf{x}))$. For larger values of d population count can be done in time $O(\lg^* d)$ (by iterating the HAKMEM algorithm).

From a mathematical point of view, the entropy for one split along the median is certainly maximal. But even for simplified Mixed Gaussian Distributions (2D, shifted mean, same deviation but same amplitude) entropies can only be approximated.

The number of regions distinguishable by the BVM rises exponentially with dimensionality. For dimensionality d we have 2^d possible regions. As we split the data component-wise, the BVM corresponds to a static decision tree that has depth d and that is independent of the number of elements. An explicit construction of such a tree, however, is not required. In contrast to Support Vector Machines, the binarization automatically normalizes the input data to the unit hypercube.

9.3.2 Case Studies

One question is if, and when the binary kernel is better than a linear one. If we assign each vector $\{0,1\}^n$ with even population count to class 1 and each vector $\{0,1\}^n$ with odd population count to class 2, then we generalize the XOR problem into higher dimension (the minimal Hamming distance of two elements in one class is 2). This is clearly not linearly separable, but the BVM can find a perfect classification.

This clearly is a best-case scenario, but we can argue that linearly non-separable but binary separable examples are common in practice. One reason for this is that many classes underlie the principles of differentiation and composition. Let us assume for example a set of images of noses, taken either from a left angle or from a frontal perspective. Although left noses are visually and numerically similar to each other, they differ strongly from frontal noses and occupy a separate region in feature space. The combined class *nose*, however, is activated by features from both differentiations. The dispersed placement of multiple clusters of sub-classes in feature space can easily create non-linearly separable situations, especially in multi-class problems. Moreover, real-world data

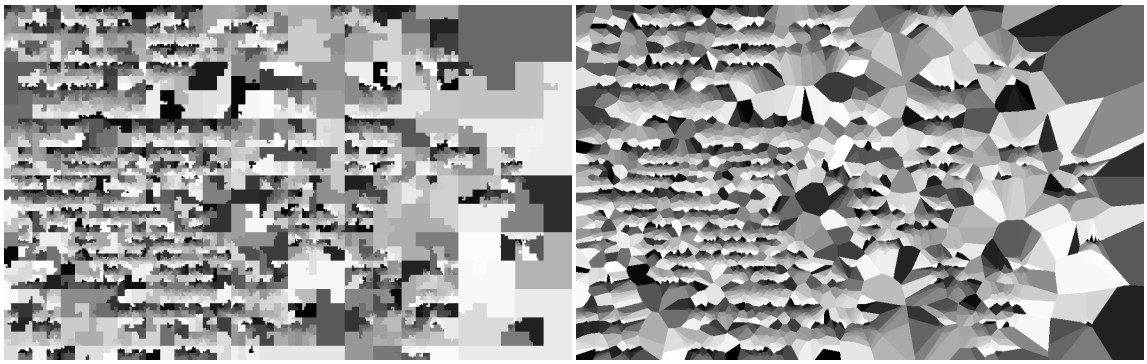


Figure 9.3: Results of Hilbert and kd tree computation.

often comes from independent or principal component analyses. As a result, feature distributions for different sub-classes tend to be iso-oriented (aligned with the Cartesian coordinate axis).

One worst-case scenario for the BVM in two dimensions is a checkerboard layout of two classes because after two orthogonal cuts all fields of the checker-board that fall into the same quadrant are represented by the same bitvector and with it the same class. For this theoretical setting, the SVM calls for several support vectors and likely an overfitting of the data. If the number of Gaussian kernels is small and the dimension is high, the BVM has good chances to find a discriminative partitioning. As a result, feature binarization preserves high selectivity and lifts the curse of dimensionality.

Even though entire orthants collapse to single data points on the hypercube, the Gaussian weighting of the binary vectors still preserves at least some geometrical meaning. The weights for a specific class are propagated nonlinearly from one orthant to another via shared hyperedges. For shared hyperedges the Hamming distance of the associated bitvectors is 1.

9.4 Experiments

For the nearest neighbor search, we take an image of size 1280×800 with about five thousand keypoints (see Figure 9.3), while obtaining a speedup of about 9 for the Hilbert curve compared to a *kd*-tree.

Operation	Time
Preparing Hilbert	190,320 μs
Hilbert Nearest Neighbor	26,364 μs
per pixel	0.0257461 μs

Operation	Time
Building *kd* Tree	1,872 μs
kd Tree Nearest Neighbor	237,744 μs
per pixel	0.232172 μs

To validate the bitvector machine experimentally, we produced training and test data for two and five-class problems in a random process with defined statistical properties (see Figure 9.4). For each class we realised a sampling of a *mixture of Gaussians*. The mean $\mu_i, i = 1,2,\ldots,$ of each multivariate Gaussian

$$p_i(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\mu_i)^\top \Sigma^{-1}(\mathbf{x}-\mu_i)\right) / (2\pi)^{d/2}|\Sigma|^{1/2} \tag{9.4}$$

is placed in the unit hypercube. The bandwidth is set globally in the main diagonal of the covariance matrix Σ for all Gaussians. The number of Gaussians is set individually for every experiment.

The maximum likelihood estimate (Bayes) of the Mixed Gaussian distribution is used as the ground truth to which a Support Vector Machine and the above method are compared. Linear kernels are used to detect linearly separable situations.

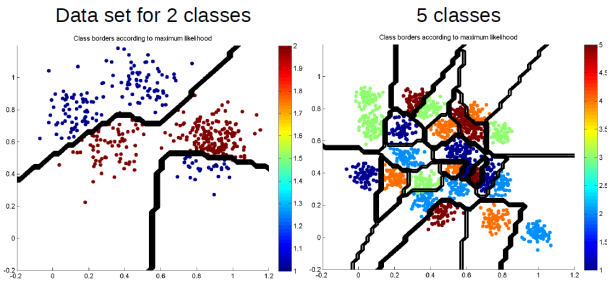


Figure 9.4: Datasets for two- and five-class problems together with maximum likelihood class borders.

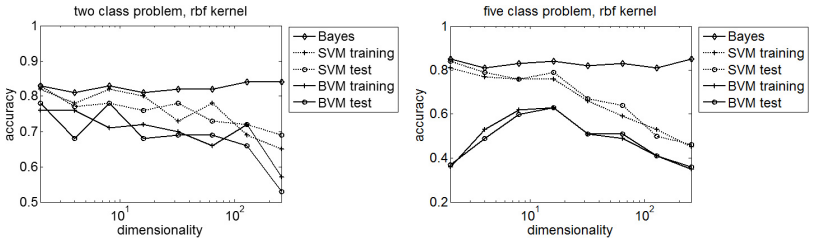


Figure 9.5: First scenario: Accuracy for the SVM and BVM using the RBF kernel

The difficulty of the created problems is controlled by adjusting the bandwidth of the Gaussians for a specific accuracy of the maximum likelihood estimator. If we keep the number of Gaussians fixed and increase the dimensionality, we must to increase the bandwidth, too. Otherwise, the overlap between the Gaussians decreases and the difficulty of the problem changes. The bandwidth of the kernel is therefore a parameter. It exhibits a strong influence on the resulting estimate. We therefore set it manually.

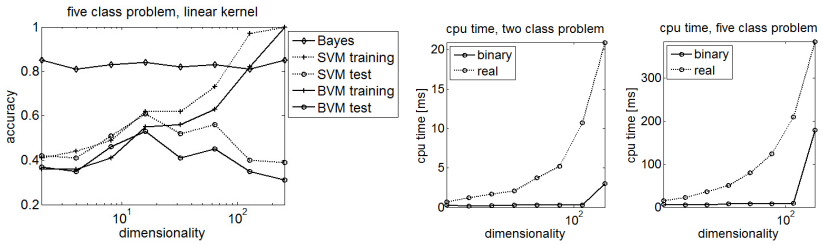


Figure 9.6: Accuracy for the five-class problem of scenario 1 using a linear kernel instead of the RBF kernel (left), and CPU time measured for the classification of the data sets (right).

In the first scenario we have synthetic experiment. As a parameter we choose the bandwidth to provide accuracy values to be in the range $[0.8, 0.85]$. For determining the classification accuracy (on the training and test sets) we conducted two experiments, one for a two-class problem and one for a five-class problem, both with rising dimension (dimensionalities 2, 4, 8, $\ldots$, 256). The two-class problem is modeled by three Gaussians per class at uniformly distributed random positions in the unit cube. With five Gaussians per class, the five-class problem is more complex. Each Gaussian is sampled 70 times. The data set is randomised and split into equally sized training and test sets. The parameters C (a weighting of the slack variables in the optimized function) and γ of the RBF kernel of LIBSVM are optimized using Python scripts.

The plots in Figure 9.5 show that the BVM approximates the original SVM surprisingly well, given the limitation that the BVM does not represent any gradual or continuous feature values. For larger dimensions, the accuracy drops for both methods because the number of training samples is small.

This is validated in another experiment, where we trained a linear kernel without cross-validation for the SVM, and for completeness also for the BVM. In Figure 9.6 (left) we observe that for high-dimensional problems the accuracies in training and test diverge. The beginning of this overfitting corresponds to the decrease of the accuracy of the SVM using the RBF kernel for more than 32 dimensions. For lower dimensions we see advantages for the BVM (RBF kernel, Figure 9.5) over the SVM with linear kernel (Figure 9.6, left).

In summary the accuracy of all classifiers is limited, so the experiment may be overly complex. Even the linear kernel might be competitive if additional application constraints are considered.

In matters of CPU time we see a drastic decrease in computation time for the BVM. Figure 9.6 (right) shows that we obtain a huge speed-up that additionally increases with the dimension of the problem. The maximum increase of performance was a factor of 32 for the two-class problem in 128 dimensions. The drop of performance at $d = 256$ can be explained by the increased word length. This means that all word level operations on a 64-bit machine have to be executed four times.

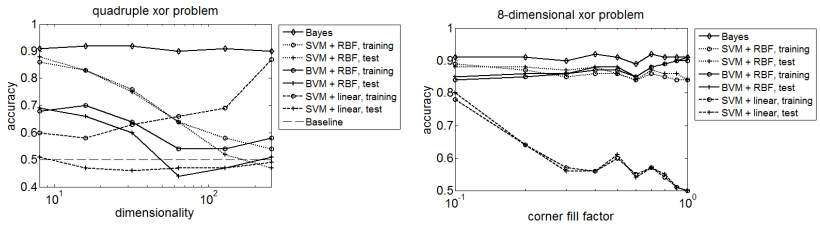


Figure 9.7: Accuracy for a two class problem including four planar xor problems (left), and for a non-linear, eight-dimensional class arrangement of increasing complexity (right).

The second scenario represents the above-mentioned high-dimensional XOR problem but with noisy data. To this end, we center the Gaussians of the Gaussian Mixture at the corners of the unit hypercube and assign class labels to the Gaussians as described above. The bandwidth is adjusted so that the Bayes classifier achieves an accuracy of 90%–92% using the known distribution.

Feature vector	SIFT		SIFT, absolutely oriented					
Foreground samples per class	20		20		125		250	
Background samples	2000		2000		1125		2250	
Classifier	Training	Test	Training	Test	Training	Test	Training	Test
LibSVM, RBF	73.4	74.0	79.9	79.1	84.7	85.6	86.8	87.9
BVM, RBF	67.2	67.3	74.0	75.0	82.4	83.2	84.2	85.1
LibSVM, linear	99.1	60.7	97.7	70.5	94.6	77.4	90.9	79.6

Table 9.1: Accuracy [%] for SIFT data sets of different size and class distribution. There are 15 foreground classes and one background class. The samples are randomised and split into equally sized training and test sets.

Figure 9.7 (left) shows the results of the discussed method and the SVM using the RBF and linear kernel. The Gaussian Mixture consists of four planar XOR problems placed at random sides of the unit hypercube of dimensionality 8, $16, \ldots, 256$). A planar XOR problem consists of four Gaussians at the corners of a square with diagonally different class labels. Each Gaussian is sampled 70 times. The best results can be seen for the SVM using the RBF kernel. The output of the BVM follows the SVM at a lower level. The linear classifier fails completely because it cannot represent the non-linear class borders.

Figure 9.7 (right) shows the results for an eight-dimensional XOR arrangement of varying sparseness. For this distribution, Gaussians are randomly placed in a certain percentage of all corners and sampled 100 times each. For a sparse filling of 10%, the distribution still seems linearly separable. However, with increasing density the linear classifier quickly approaches random, whereas the BVM improves gradually and finally outperforms the SVM with RBF kernel.

The third scenario is a computer vision task, where SIFT descriptors are classified into 16 classes (15 classes representing different parts of a face plus one background class). The binarization has 128 bits, so that each vector consists of two 64-bit words.

No	BVM + 16BITPC	BVM + NP	LIBSVM
1	2.16	1.00	48.36
2	2.16	1.00	48.27
3	2.14	1.01	48.59

	Kernel computation using all support vectors		
No	LIBSVM	BVM + 16BITPC	BVM + NP
1	39.44	4.01	2.84
2	39.45	4.01	2.81
3	39.71	4.00	2.71
	only support vectors with $\beta \neq 0$		
1		3.46	2.27
2		3.45	2.28
3		3.46	2.28

Table 9.2: Evaluation time in seconds for 116,072,232 kernel computations (left), Time in seconds for classifying 16,788 vectors in 15 classes, 6,914 support vectors.

We evaluate the effectiveness of population counting on the machine and study the speed-ups obtained for sole kernel evaluations (Table 9.2, left) and whole vector classifications (Table 9.2, right). We compare the LIBSVM implementation with the BVM in two settings, one with a precomputed 16-bit population count lookup-table (2^{16} entries, cf. Program 9.1), one with the native population count (`__builtin_popcountll`). We run three examples for each setting to show that the variance in the running times is small. The table documents a speed-up factor 48 in the 116 million kernel comparisons and a factor 17 improvement for the entire classification process. Further speed-ups might be achieved by using a one-versus-all strategy for multi-class problems instead of the one-versus-one strategy of LIBSVM.

Program 9.1: Non-native bit count with precomputed lookup tables.

```
int BitCount[65536], BitCount_IsUndef = 1;
int BitCount(uint64_t a) { return BitCount16(a); }
void CreateBitCountLUT() {
  if (BitCount_IsUndef) {
    uint64_t n = 65536;
    while(n-- > 0) BitCount[n] = BitCount4(n);
    BitCount_IsUndef = 0; }}
int BitCount4(uint64_t a) {
  static int BitCount[16] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
  int Count = 0;
  while(a) {
    uint64_t LowestNibble = a & 0x000000000000000fLLU;
    Count += BitCount[LowestNibble]; a = a >> 4; }
  return Count; }
int BitCount16(uint64_t a) {
  CreateBitCountLUT();
  uint64_t w0 = a & 0x000000000000ffffLLU, w1 = a & 0x00000000ffff0000LLU;
  uint64_t w2 = a & 0x0000ffff00000000LLU, w3 = a & 0xffff000000000000LLU;
  return BitCount[w0] + BitCount[w1 >> 16] + BitCount[w2 >> 32] + BitCount[w3 >> 48]; }
```

The difference in CPU time between the sole kernel computation and the whole classification indicates a strong influence of the code analysis and generation of the compiler, since the number of kernel computations was equal in both experiments, and the optimization flags too.

Figure 9.1 shows for different SIFT variations and different class distributions that despite the high speed-up the accuracy of the BVM is close to that of the SVM. The BVM also always outperforms the linear classifier.

9.5 Summary

Deep learning boosted the accuracy of many computer-vision tasks, and by visualizing the hidden units of a convolutional neural network, we get an impression of what might be general concepts that have been learnt. Nonetheless, improved support vector machines and nearest neighbor search are equally important, and often better suited to a specific task. They are generally more efficient to train, work with non-trivial kernel functions, and lead to higher accuracy values for difficult vision tasks.

Established *descriptors* are used to map the bitmaps to features that are then being used for both training and classification. In short, processing large amounts of images crucially depends on the availability of efficient machine learning algorithms.

The Hilbert curve is used to compute fast approximate solutions of the all nearest neighbor problem and the k-nearest neighbor problem in the image plane. The method has a precomputation time of $O(n)$ for adapting to a fixed image size. Depending on the problem, an additional precomputation time of $O(C+n)$ (all nearest neighbors) or $O(n \lg\lg C)$ (k-nearest neighbors) is needed to adapt to a certain set of key points. The query time is then $O(1)$ or $O(\lg\lg C + k)$, respectively. This is an advantage over a balanced tree (with runtime $\lg n$) if $n > \lg C$ (the latter being a small number for common image formats). The experiments show that the method yields a compact and visually meaningful approximation of the Voronoi diagram in the image plane, which is sufficient for many applications. For 50% of the queries, the method yields the exact result. In a practical example of finding all nearest neighbors of a set of keypoints, the method was 9–18 times faster than a kd-tree.

The BMV is a machine learning algorithm whose advantages result from a dramatic simplification of the input data. Discretization certainly has limits in the accuracy of not iso-oriented class distributions. We argue, however, that iso-oriented classes with non-linearly separable sub-classes occur in many pattern-recognition tasks.

The rising number of positive results in compiling feature vectors into bitvectors prior to the learning process shows that the effectiveness and efficiency of the binarization in the BVM is an exciting phenomenon. Especially for a growing number of dimensions, where we lack a visual interpretation and where unexpected results like the curse of dimensionality have been measured, research might have concentrated on aspects that do not discriminate well. Even though binarization reduces the information in the input considerably, literature review shows that the results are often of acceptable quality, sometimes even better than the original unabstracted input.

The experiments on synthetic and real data show that the accuracy of the BVM approximates (and in special cases exceeds) the accuracy of a Support Vector Machine with RBF kernel but is up to 48 times faster in the kernel computation and up to 32 times faster in classification. These results confirm the theoretical proof of the efficiency of classification and binarization. Compared to a linear classifier, the accuracy of the BVM is usually higher or equal. Furthermore, the BVM can model non-linearly separable problems where the linear classifier fails. Together with an improved empirical basis we provided insights that increase the understanding of when and why the approach works well, especially for large feature vectors. The BVM, therefore, allows for a welcome new trade-off between accuracy and running time for non-linear problems.

We have extended the initial findings on improved classification time to different ends: improved learning, new kernel functions, support of many bits. In particular we provided: flexible state vector sizes, rather than a fixed one imposed by the computer word length; accelerated training rather than improved classification time to cover the entire machine learning process; different kernel functions rather than Gaussian ones, to have full flexibility in their usage; and multi-bit projections rather than single-bit binarization for an improved time-accuracy trade-off. A library implementation smoothly extends and adapts to the LibSVM interfaces. The adaptation yields the support of a new *bitarray* data type.

9.6 Bibliographic Notes

One recent solution to the nearest neighbor problem is Full Delaunay Hierarchies [51] Its main difference to an ordinary Delaunay triangulation is that it additionally records all edges that have been used during the incremental construction. Using a randomized incremental construction of the Delaunay triangulation, the approach uses $O(n \lg n)$ expected time for building the data structure with an expected number of $O(n)$ edges. The expected number of nodes traversed for finding the nearest neighbor by a simple greedy algorithm yields an expected query time of $O(\lg n)$. Differently from many other approaches, an additional search structure is no longer needed. HAKMEM has been proposed by David Eppstein.

Machine learning with SVMs by [630] is one of the most successful approaches for many computer science applications [141]. Recent research indicates advantages for a mild weighting of distant outliers. A penalty for small or large positive outliers leads to mixed results depending on the application [467].

The number of support vectors learned affects training and classification speed. Bordes et al. [63] assume that not all training samples are equally relevant for the resulting model. Their LASVM implementation achieves a significant speedup during training by using an online approach, where the margin is determined from a small subset of the input data. Important modeling decisions can be reached even without considering the class label of a data point.

Tsang et al. [623] propose a radical simplification of the model by approximating the convex hull of the training data by an enclosing ball in the high-dimensional feature space. The allowed error of the approximation is a parameter of the algorithm. The method works iteratively to find the center of the ball.

LIBSVM is one of the most frequently used machine learning libraries. It is included in many different data mining environments like RapidMinder. There are also fast SVM implementations with linear kernels (LIBSVM) e.g., LIBLINEAR and Leon Bottou's [71] stochastic gradient descent SVM. However, for complex data where single classes comprise multiple distant sub-clusters, non-linear kernels achieve a higher accuracy.

Supervised machine learning is well described by [604] Kernel methods that can be found in work by [141, 561]. As kernel functions can be complex, the application of SVMs is large and kernels can be designed to cover simple regression to neural network approaches [560] and time series [310].

Linear-time $O(n)$ median selection algorithms are e.g., presented by [59]. Mixed Gaussian distributions can be found in [476, 511]. The *kd*-tree has been studied by [44] in computational geometry [150].

The curse of dimensionality [47] can be reduced by choosing a smaller norm than the Euclidian one [5], but taking high roots is numerically difficult, too. SIFT goes back to work by [460] and SURF to work by [39]. Feature vector does not suffer from binarization [597], whereas the original SIFT representation does. In some cases, the relative error rates even decreased [598]. Silverman indicates a dependency of the bandwidth on the root of the dimensionality [578].

There is a considerable progress in generating AI art via text-to-image and image-to-image with tools like OpenAI's DALL-E [297].

Chapter 10
Navigation Data

Urban mobility planning depends largely on the presence of good navigation data. Vector maps are not always available for many areas – especially for many of the third world countries. On the other hand, good paper maps collected by the city authorities are widely available. A solution is the *collaborative map generation* process that allows people to share the collected GPS traces. Nevertheless, the integration of these GPS traces is itself a challenge and requires a good *base map*.

We first consider a generic approach for map generation based on GPS traces especially for unknown terrains. The system uses AI algorithms to infer the road geometry.

Given a set of GPS traces, this input is refined by filtering and partitioning algorithms. For constructing the graph, clustering algorithms are applied that allow us to incrementally generate a road map, annotated with travel time information. The processing of the input data relies on a subdivision of the world into tiles. A flexible map viewer is provided to navigate through the hierarchically organized content of the database. The system provides the export of maps to serve, e.g., as the input for a routing server, which in turn can be queried via TCP/IP for shortest and quickest routes.

Next, we present a method to extract calibrated road topology from raster maps to provide such a *base map* for collaborative map generation process. The following approach takes a bitmap and uses different graphics filters to infer the road geometry. We introduce an aggregation algorithm that extracts the actual vectorized road fragments and constructs a graph of road network. We evaluate the algorithm on real raster maps collected from the city authorities of Dortmund, Germany. We also report the integration of the approach into SUMO, a state-of-the-art traffic simulation tool for urban mobility.

10.1 Introduction

Most[1] available digital maps are expensive to produce and update, since exhibiting and processing road information e.g., by surveying methods or by digitizing satellite images, is very costly. Maps are likely to be inaccurate and to contain systematic errors in the input sources or inference procedures. It is costly to keep map information up to date, since road geometry changes over time. In some regions of the world, digital maps are not available at all. Maps contain information on road classes and travel distances only, which is often not sufficient to infer travel time. Moreover, it is not possible to request a quickest path from a source to a destination during a specific time period.

With the industrial emergence of low-cost positioning systems and by the accelerated development of handheld devices and mobile phones, integrated data gathering and processing to assist personal navigation becomes

[1] This chapter is based on joint work with Stefan Schrödl, René Bruntrup, Stefan Edelkamp, Shahid Jabbar, Björn Scholz, Maik Drozdzynski, Andreas Gaubatz, and Miguel Liebe. It puts together and improves the work from [220, 87, 171].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_10

Figure 10.1: Incremental map generation.

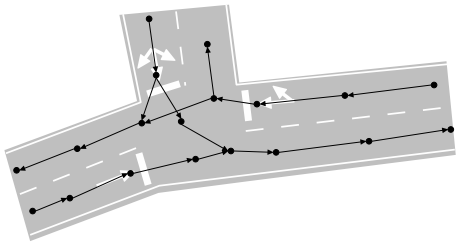


Figure 10.2: Map graph representation.

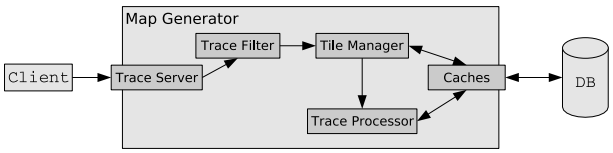


Figure 10.3: Data flow in the map generator tool.

feasible at a very large scale. In many PDAs or on-board units, GPS functionality is already available, suggesting the automated creation or adaption of a map based on GPS data.

In this text we present a client-server architecture that takes GPS data as input to incrementally *learn* (see Figure 10.1) a map using AI clustering technology. The input to be processed consists of recorded GPS traces in form of a standard ASCII stream, which is supported by almost all existing GPS devices. The output is a road map in form of a directed graph with nodes and edges annotated with travel time information. During the processing of a trace map, nodes are created at regular distances (see Figure 10.2).

Nodes and edges are stored in a database. To obtain a certain locality we chose to subdivide the world into tiles and store the nodes and edges inside the corresponding tile. Each tile covers an area of $1/100 \times 1/100$ degrees in geodetic coordinate system.

The traces are processed in parallel, i.e., we allow different trace processing threads to run concurrently. Most important, parallel access to the map database has a large potential to accelerate the runtime behavior on the server side. The data flow within the architecture is shown in Figure 10.3.

First, we consider the pre-processing of incoming GPS data, namely the *trace server*, *trace filter* and *tile manager* modules. Then, we present the algorithmic details of the core clustering algorithm in the *trace processor* that incrementally merges a new trace to the already inferred map. Afterwards, we look at the performance and accuracy of the algorithm. For map administration, we introduce a hierarchical navigation front-end that allows us to scale gradually from a global to a detail view of the map itself and the incoming traces. We explain the system's I/O architecture and how it is integrated in a client-server navigation system with hand-held access.

10.2 Map Generation

Pre-processing

In order to organize an incremental extension of the map, we first have to parse and filter the GPS traces.

The *trace server* takes a GPS trace as the input. It listens to an open port and accepts connections. The connection receives GPS traces in NMEA 0183 Standard and is closed after the data has been sent. The relevant data for map generation (latitude, longitude, altitude, absolute point in time) is extracted from the parsed GPS data items and packed into an internal data structure for further pre-processing.

The *trace filter* was realized to improve the quality of GPS traces. The main task of the filter is to detect and eliminate outliers caused by the inaccuracies of the GPS system. We detect outliers by thresholding absolute velocity and acceleration on the trace. The filter also detects gaps in between two GPS points, e.g., caused by distorted signal information of the GPS receiver. That time delay between two received data items is bounded by a threshold. If the threshold is exceeded that trace may be split. This parameter must be chosen with care. If the threshold is too small, existing tunnels may no longer be detected. To avoid such flawed behavior, we will have to combine several heuristics.

The *tile manager* module is responsible for the maintenance and propagation of GPS traces. If a new filtered trace is present for subsequent processing, a sequence of actions to integrate the trace into the map is executed: a trace is selected, the area, i.e., the tile region, on which the trace is located, is reserved and a new *trace processor* is created and started to integrate the trace. Finally, after a trace processor has finished the reserved tiles are released.

I/O and Communication

The GPS trace map is stored either in a file-based database or externally in an *ODBC* connected database. The file database was implemented for an easier program setup. For production systems the external database is recommended. ODBC provides the same functionality on all supported operating systems and supports changing the database underneath. The implementation in MySQL can be altered to other databases such as PostgreSQL without any changes to the source code.

The database contains just one table that consists of an ID and a further field for maintaining variable-length binary data. In the data field the information encoding a tile is stored in binary form.

The *cache* is the connection between the database and the other program fragments. It maintains the mapping of the data from main to secondary memory. Casually, it allows us to reduce the slow database accesses.

For internal communication and communication with the GUI we chose a *publish-subscribe system*, a simple and generic interface to facilitate state synchronization between cooperating components. The design pattern extends the *observer* pattern, providing a mechanism for decoupling dependencies between applications. It has been extended to a *publish-subscribe channel* to provide a way to broadcast events to all receivers that have subscribed to a specific topic.

More complicated constructions are possible: as we allow a subscriber to also be a publisher, processing chains can be realized. For example, the cache values of different publishers could be received and their average be published.

Observations and Experiments

We took a small trace collection of real-world traces.

We measured the performance in two different tests. In the first test we compared two different computers with a cache size that was big enough to store nearly all data. The values should reflect mostly the scalability of the core algorithm. The second test compares the whole system performance using different cache sizes on one of the computers.

In the first test all available traces where send to the server as fast as possible using a simple shell script and *netcat*. We applied two scenarios: In the *creation phase* we used an empty database and cache. This shows how fast an initial map can be computed. In the *update phase* the database contained all roads, and the cache was already filled. This is probably a more realistic scenario for a production server. We ran the performance tests on two different linux machines. Table 10.1 shows the aggregated user and system values, as the unix *time* program shows them. These values only reflect the CPU time the program used. As the program was able to push the CPU usage to above 90% the real times where only slightly higher all the time. As we can see the core algorithm scales very well with the CPU speed. As expected, updating is much slower, because much more points have to be clustered together. In both cases and on both computers the speed is still high enough to process the data that an average car produces per hour in less than a second.

As a second test we measured the whole system performance and the influence of the cache, by running the tests with different cache sizes. This tests were only made on the Duron. For this test a dataset was sent to the server three times and the complete processing time including writeback of the data was measured. Opposed to the first test we used a local mysql database. This is slower, but more realistic. The cache used a simple FIFO logic. Table 10.2 shows that in this setup the cache size is very important, even with the local database. With the database on another computer the difference would probably be even higher.

Table 10.1: Computer comparison

	Computer1	Computer2
Creation	117.45	236.55
Update	87.19	161.91

Table 10.2: Cache comparison

Size (KByte)	1,000	5,000	10,000	50,000
Speed (km/s)	36.89	39.29	42.47	49.13

There are sources for inaccuracies in the system. On the one hand there is the inaccuracy of the GPS system itself and on the other the algorithm may misinterpret the incoming data.

The US Army terminated the intentional degradation of the GPS signals (called SA) on May 1, 2000. This augments the accuracy to 15 meters at least. If we assume a symmetric distribution of the error values, a merge of nearby points belonging to the same road can even raise the accuracy. Therefore we can expect GPS data to be fairly exact if enough test data is available.

As already mentioned in the algorithm's description, the correct evaluation of road connections is crucial for accurate maps. The first and probably most important result was that the generated map matched a conventional road map in most cases, which we verified using the *map viewer* described in the next section. In the current implementation the algorithm can correctly determine most road structures. Problems occur with narrow structures, such as close crossings and mini roundabouts.

10.3 Map Administration

To test and observe the server, and to navigate through the database, a flexible graphical user interface is mandatory. The information on the actual state of the queued data being processed and the map being generated, i.e., the contents of the database, are accessible to the server operator.

For a smooth working of the program, it is essential to recognize communication problems and bottlenecks. Therefore, error and status information, e.g., the number of working *trace processors* or the number of pending GPS traces, are visualized. As the representation of such statistical data in pure textual form is not feasible, the graphical user interface provides a concise representation of all messages and status variables of the server.

Here, we concentrate on displaying the map using a graphical front-end based on a world-wide hierarchical navigation. As we already mentioned the complexity of road inference mechanism itself, the tool was a necessary aid for debugging.

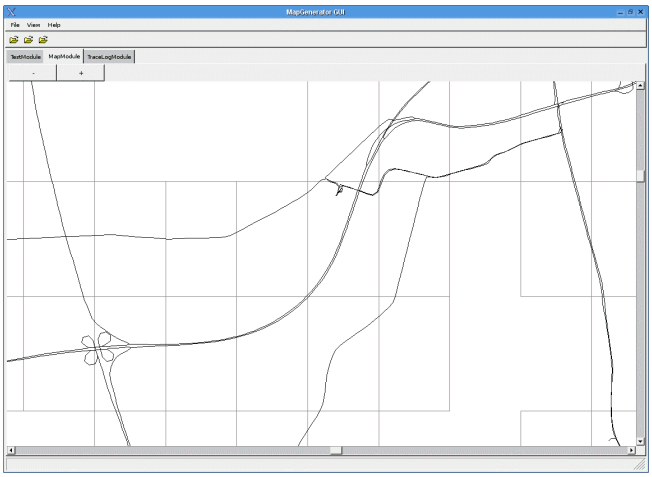


Figure 10.4: A view of the map in the database.

The *map viewer* displays the actual map as generated by the server. Since only a read access to the database is needed, the viewer can be used without any problem on a running server.

Figure 10.4 shows a part of the map in a moderate zoom factor. We see highway crossing and parts of a city (upper right corner). We support the facility to zoom the map from a global view down to the visualization of single road elements.

As the server runs on GPS data, we use a UTM transformation to project the 3D world to the 2D map representation window. For displaying the entire world, we have to cope with $36,000 \times 18,000 = 648,000,000$ tiles. Loading that many tiles into main memory is infeasible and even a test of road existence on each tile would be too time-consuming. Therefore, we first load the IDs of all used tiles into main memory. With an expected usage of 1% of all tiles we have an accumulated data volume of about 25 MB for the 4 Byte IDs only. Assuming a fast networ k that is required to use the viewer anyway, this performance requirement is acceptable.

The list of used tiles is compacted into 2-bit vectors to allow a faster access to active tiles and to save memory. In a coarse scale the usage of larger tile blocks is queried. Only those meta blocks are shown that contain any road information. Only in the fine-grained display modus we access the database to display the data that is contained in the tiles.

The access to the database is performed in the background to allow displaying without any distortion. The data of the tiles is displayed once it is read from the database into the cache. Using this implementation, the user can navigate quickly to all selected magnifications of the world map as long as the caching policies to the database

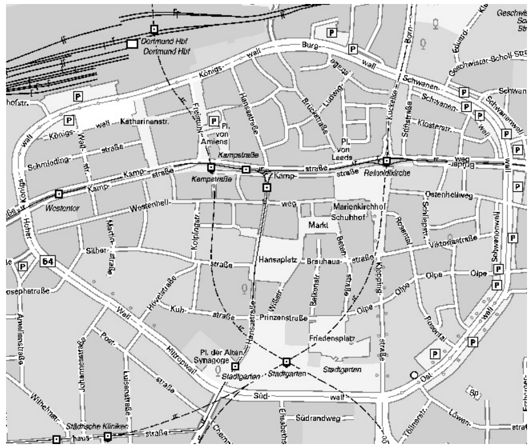


Figure 10.5: A fragment from a topographic map.

are efficient enough to predict the user's behavior. In practice, we observed no bottleneck in the graphics for navigation or animation.

The inferred map can be exported in different formats for use in different applications. The routing module is called via TCP/IP and computes the shortest and quickest route between two query points. The TCP/IP connection is accessible for handheld devices. The project group implemented and tested a system on a PDA. The handheld application protocols GPS traces, visualizes traces, and accesses the routing server. For the user's orientation, a topographic map is also included, which can be calibrated with the input GPS signal. The presented approach has been successfully integrated into this route planning system.

10.4 Extraction of Road Surfaces

The raster map we started from contains a set of pixels. Each pixel at position (i, j) carries its color information in form of an RGB triple. Fortunately, the set of topographic maps we considered do not contain much noise, such that the streets are easily discriminated by their color value (see Figure 10.5). We allow the user to select the color value for road extraction. Next, we set all the information that belongs to the street network to black, and all other information to white. Using this simple filter yields some subtle problems: in the maps we have black letters for street and city names and further urban information; moreover, all railway tracks are drawn in black.

The solution to define black as the color of the road includes railway tracks. On the other hand, not using black would give rise to many white holes in the road infrastructure due to the street names. Moreover, as parking and playgrounds are white in the input, selecting this discriminating color also has its drawbacks. Subsequently, the current extraction process is not perfect and needs some manual post-processing.

For automated post-processing we implemented six morphological filters: *Erosion*, *Dilatation*, *Morphological opening*, *Morphological closing*, *Gap closing*, and *Fragment elimination*. We illustrate the working of these graphical filters in the following using a running example. The mathematical basis for these filters is set theory and they are termed morphological filters because they work on the *shape* of the image.

Erosion In order to remove the city and street names from the map, we employ an erosion operation. An erosion operation is defined with respect to a *structuring element* called mask. Intuitively, it works like a net with holes in the shape of a mask. All the elements of the image that can pass through the holes disappear.

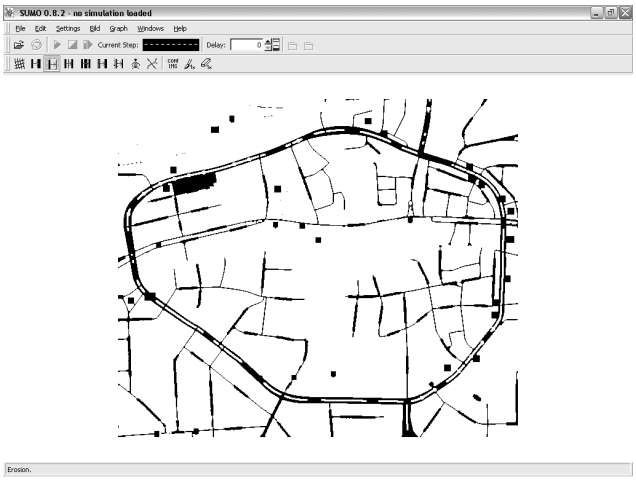


Figure 10.6: Result of erosion

For two integer matrices A and B, the *erosion* of A with respect to B is defined as $A \ominus B = \{z \mid (B)_z \subseteq A\}$. Therefore, an erosion of A with respect to the mask B consists of all points z, such that if B is shifted with its center point at z, it remains in A.

The form and size of the mask is crucial for the results of erosion. To avoid distortion of the road elements, a symmetric mask is required. As pixels are deleted by erosion (starting at the fringes of the road surfaces), the mask should not be selected to be too large, suggesting a (3×3)-pixel map in our case. Using erosion, a large part of the non-road elements such as railway tracks and names are eliminated. The result of erosion on the example is depicted in Figure 10.6.

Dilatation Due to the application of erosion operation, street lines might become distorted, especially at the places where they overlapped a street name. A dilatation operation is then used to smoothen these contours and to fill certain holes that might have appeared during erosion.

For two integer matrices A and B, the *dilatation* of A with respect to the mask B is defined as $A \oplus B = \{z \mid (B)_z \cap A \neq \emptyset\}$. Dilatation enlarges the number of black pixels and closes some gaps within the road surfaces. As in erosion, the choice of mask is crucial, and it should be taken care of that no two road elements are merged into one.

Morphological Opening The morphologic opening is a composite morphological operation based on both erosion and dilatation. For two integer matrices A and B, the *morphological opening* of A wrt the mask B is defined as $A \bullet B = (A \ominus B) \oplus B$. This operation helps in removing small bridges between black surfaces, smoothening of the contours and elimination of small noises.

Morphologic Closing The morphologic closing of a set A given the structuring element B is defined as $A \circ B = (A \oplus B) \ominus B$. Using this operation will also smoothen contours. In contrast to a morphologic opening, small bridges between black surfaces are strengthened and small gaps and indentations are filled.

Gap Closing A problem that cannot be solved with the dilatation alone is the elimination of gaps within large road elements such as inter-state highways.

Several dilatations can close those gaps but may lead to a merging of different road fragments. A solution to the problem is a specialized algorithm *gap closing*, which tests for each white pixel if it has more than $n \in \{1,2,\ldots,8\}$ black neighbor pixels. The number n is to be provided by the user, with $n = 5$ as a feasible choice.

Fragment Elimination Even after applying the above-mentioned filters, some of the railway tracks and other map symbols remain on the map as isolated fragments. Such fragments are searched and removed in the *fragment elimination* phase.

We iterate through every pixel of the bitmap. If we find a black pixel x, the neighborhood of x is analyzed. We exploit the property that route fragments accumulate to continuous chains of black pixel. If all surrounding pixels of x are white, we assume x to be isolated from the other black pixels and color it white. In the method an odd integer $i > 3$ is taken as a parameter, which denotes the size of the square that is drawn around x. In a loop i is increased successively. If all the border pixels of the square are colored white, the iteration is stopped, coloring all internal pixels also white. If this is not the case, the square is enlarged until the upper bound is met. Corner pixels of the bitmap must be considered in a refined case study.

10.5 Road Skeleton Computation

After the road surfaces have been clearly extracted using the application of filters and some additional manual refinements, the skeleton has to be computed as a basis from which the street graph is generated. Roughly speaking, the skeleton of a pixel map is a set of thin curves, denoting the centerlines of the black surfaces. With respect to the road geometry, these are the centerline of the streets to look at. Using the mathematical notation of morphological arguments the skeleton of A is defined as $S(A) = \bigcup_{k=0}^{K} S_k(A)$ with $S_k(A) = (A \ominus kB) - (A \ominus kB) \circ B$, where B is the structuring element and $(A \ominus kB) = (\ldots(A \underbrace{\ominus B) \ominus B) \ominus B) \ominus \ldots) \ominus B}_{k \text{ times}})$ Moreover, K is the last iteration before A is the empty set, i.e., $K = \max\{k \mid (A \ominus kB) \neq \emptyset\}$.

10.5.1 Skeleton Generation

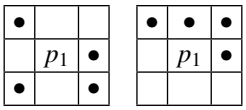
A first implementation of the classical definition of the skeleton has yielded the aimed result. But a skeleton generated by using the above-mentioned definition is not guaranteed to be connected and may easily break down into pieces especially at crossings. This can make graph generation infeasible.

The medial axis transformation (MAT) of a region R with fringe G is defined as follows. For every pixel p in R, we search for a nearest neighbor in G. If there are more than one neighbor, then it belongs to the middle axis, the skeleton. An illustrative interpretation of the MAT is a fire in the Savannah, with dry grass that is set into flames at its fringe. The MAT corresponds to all positions that are reached by at least two fire frontiers. A direct conversion of this idea leads to inefficient algorithms since the computation for distances from all interior points to all fringe pixels is involved.

The binary bitmap is processed in two stages that are successively altered until no more changes are to be observed. Let the considered pixel be arranged as follows.

p_9	p_2	p_3
p_8	p_1	p_4
p_7	p_6	p_5

In the first stage, all black pixels are marked that satisfy the following conditions: $2 \leq Sum(p_1) \leq 6$, $Trans(p_1) = 1$, $p_2 \wedge p_4 \wedge p_6 = 0$, and $p_4 \wedge p_6 \wedge p_8 = 0$, where $Sum(p_1) = p_2 + p_3 + \ldots + p_9$ and $Trans(p_1)$ denotes the number of 0/1 transitions in the sequence $\langle p_2, p_3, \ldots, p_9, p_2 \rangle$. In the following, two cases of the pixel's positions are shown. On the left, we have $Sum(p_1) = 4$ and $Trans(p_1) = 3$, while on the right, we have $Sum(p_1) = 4$ and $Trans(p_1) = 1$ as required for the second condition.



Condition 1 is violated if p_1 has one, seven or eight black neighbors. In case of one neighbor p_1 cannot be deleted as it is at the end of a pixel chain. If black pixels with seven or eight neighbors are deleted, the region will quickly become sparse. Condition 2 takes care that the skeleton is not split, so that no pixel is deleted that lies on a chain of minimal width.

If all conditions are checked, the original is deleted. The manipulated bitmap is the input for the next step.

In the next stage, all black pixels are marked if the following conditions are satisfied. $2 \leq Sum(p_1) \leq 6$, $Trans(p_1) = 1$, $p_6 \wedge p_8 \wedge p_2 = 0$, and $p_8 \wedge p_2 \wedge p_4 = 0$.

A pixel that satisfies the first two conditions and both first parity conditions is located east or south or is a north-west corner point. In this case the pixel is not in the skeleton and should be eliminated. Similarly, a pixel is eliminated that satisfies both second parity conditions is located north or west or is a south-east corner point.

In Algorithm 10.1, we see the pseudo-code form of the skeletonizing algorithm that checks the entire bitmap for the pixels that satisfy the former conditions. Note that the pixels are not deleted in the first pass and only marked, so that they do not bring any change to the next iteration. In a second pass, all marked pixels are deleted.

Algorithm 10.1: Skeleton generation.

```
Input: 2DPixelArray p; Height; Width;
repeat
  Marked ← false;
  for i ← 1 to Height do
    for j ← 1 to Width do
      if p(i, j) = 1 then
        if 2 ≤ Sum(p(i, j)) ≤ 6 and Trans(p(i, j)) = 1 and
          p(i, j − 1) = p(i + 1, j) = p(i, j + 1) = 0 AND
          p(i + 1, j) = p(i, j + 1) = p(i − 1, j) = 0 then
           Mark(p(i, j));
           Marked ← true
  for i ← 1 to Height do
    for j ← 1 to Width do
      if Mark(p(i, j)) then
        p(i, j) ← 0;
        Unmark(p(i, j))
until Marked = false
```

10.5.2 Skeleton Minimization

An immediate construction of the graph from the skeleton is problematic since the skeleton is not necessarily *maximally sparse*. A skeleton is *maximally sparse* if the elimination of a pixel having two neighbors can break down the skeleton. This condition can be used to minimize the skeleton and to identify pixels corresponding to crossings, dead-ends or to just an ordinary point. In total, we must consider $2^8 = 256$ cases. The result of the skeleton generation and minimization phase is shown in Figure 10.7.

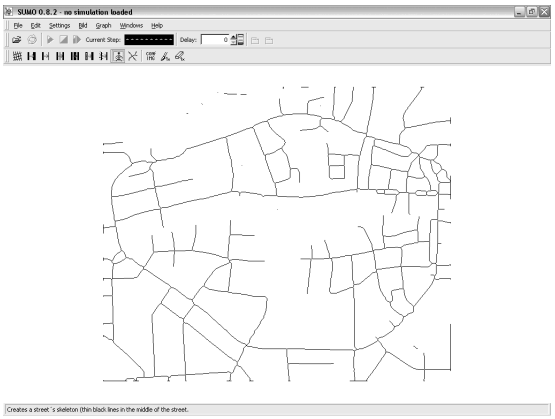


Figure 10.7: The generated skeleton.

10.6 Graph Construction

Once the skeleton is generated, a *Tracking algorithm* constructs the underlying directed graph. This algorithm is based upon the Sweep-line paradigm: the pixels are processed in a column-wise fashion. As soon as a new pixel is found, which belongs to the skeleton, a sub-routine is invoked that traverses and processes the connected component of the new pixel.

It generates nodes for all neighboring points. Furthermore, new pixels are added to the pixel chain at regular intervals (user-defined) and are connected with the previous pixels by new edges. As soon as a pixel representing a crossing is reached and converted to a node, a list of its neighbors is generated. This list is then iterated through, calling the same sub-routine recursively for every list element.

One thing that is to be especially taken care of is that the neighbors are not accessed twice. It can happen that during the traversing of the skeleton in this manner, a crossing is encountered that had been visited but not all of its neighbors had been processed. This problem can be resolved by using a flag that marks a pixel as visited and avoids re-processing when reached again.

Once a connected component has been processed completely, the sweep-line algorithm searches for another connected component in the picture. A node counter is set that counts the number of nodes that belong to a connected component. In this way we can get rid of those connected components that are too small to represent any street. These fragments may correspond to any noise that may have been left during the filtration process. The minimum size of a connected component should be provided by the user. Figure 10.8 shows the street graph constructed for the example.

The graph constructed through the *Tracking algorithm* has a large number of nodes. In case of very large raster maps, the resulting graph could be enormous in size and would not even fit into the available memory. The graph can be simplified by the help of simple observations. Using some information on the road geometry, the number of nodes that are needed to represent a street can be reduced. In many cases, we have achieved a reduction of about half the nodes without losing any significant information about the road geometry. In the following, we discuss the two simplification techniques that are used in the approach.

Removal of Redundant Nodes A node u is redundant if it lies in the middle of a straight street. Typically, such nodes have an *in-degree* = *out-degree* = 1. Note that the street must be straight, else we may lose the desired precision in road geometry. Let $u = (x_u, y_u)$, $v = (x_v, y_v)$, $w = (x_w, y_w)$ be the three consecutive nodes in the graph G. The simplification algorithm basically uses a collinearity test to find the redundant node. The node v is redundant if v lies on the line joining the nodes u and w. The collinearity test can be performed by testing if the value of the determinant

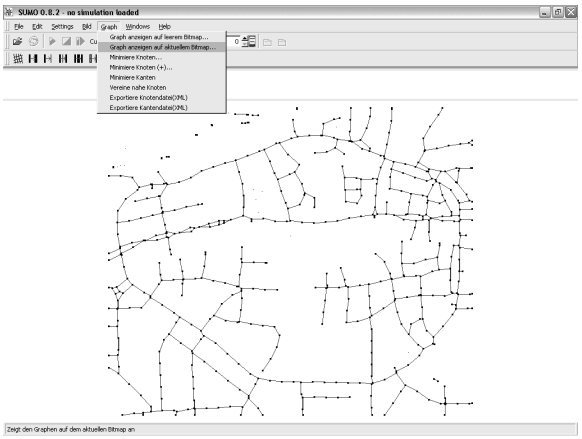


Figure 10.8: The street graph.

$$\begin{vmatrix} x_u & y_u & 1 \\ x_v & y_v & 1 \\ x_w & y_w & 1 \end{vmatrix}$$

is 0 or not.

The simplification algorithm described above can be adjusted to simplify a bit curved or wavy streets too. By introducing a tolerance parameter ε, only the nodes that lie within an ε-distance of the line joining the two neighboring nodes are removed. Increasing the value of ε can give lower number of nodes but a distortion in the road geometry as many curved streets would be approximated as straight ones.

The algorithm runs in $O(n^2)$ time and is a simplified form of the Douglas-Peucker algorithm, which was developed to reduce the number of points to represent a digitized curve from maps and photographs. It considers a simple trace (certain forms of self-intersections can also be handled) of $n+1$ points $\{p_0,\ldots,p_n\}$ in the plane that form a polygonal chain and asks for an approximating chain with fewer line segments. It is best described recursively: to approximates the chain from point p_i to p_j, the algorithm starts with segment p_ip_j. If the farthest vertex from this segment has a distance smaller than a given threshold ε, then the algorithm accepts this approximation. Otherwise, it splits the chain at this vertex and recursively approximate the two pieces. The $O(n\log n)$ algorithm takes advantage of the fact that splitting vertices are to be located on the convex hull. It has been improved to $O(n\log^* n)$, where $\log^* n = \min\{k \mid \underbrace{\log\log\cdots\log}_{k \text{ times}} n = 1\}$.

Merging of Closely Situated Nodes Another approach that can result in a simplified graph with much fewer nodes is to remove the nodes that lie very close to each other. Again, it can affect the smoothness of the streets, but when done with proper parameters the resulting graph can retain a good approximation of the road geometry and still consist of less nodes. The simplification can be done by a linear time algorithm that runs in time proportional to the number of edges. The algorithm iterates through the edge list. Let s and t denote the start and the end node of the edge e, respectively.

If for edge $e=(s,t)$ the Euclidean distance between s and t is smaller than σ, where σ is the simplification parameter, the node t is declared as a *close* node and is deleted.

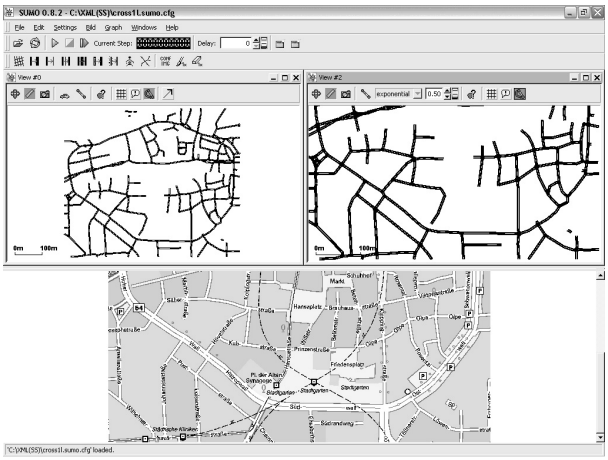


Figure 10.9: SUMO.

10.7 Integration to a Traffic Simulator

To test a utility of the approach in a different scenario, we decided to integrate the efforts with an existing traffic simulation system. SUMO is a microscopic and multi-modal traffic simulation tool. The input to SUMO is a graph in XML format. It consists of three parts: the nodes, the edges, and a third data set that consists of the test routes in the street graph. These routes are to be used by the virtual vehicles during the traffic simulation.

For the parsing of XML data, an open-source parser is used. In the following, excerpts from the XML files are shown that contain the nodes and edges information, respectively:

```
<nodes>
  <node id="0" x="0.0" y="220.0 type="priority" \>
  <node id="1" x="61.0" y="234.0 type="priority" \>
  <node id="2" x="98.0" y="232.0 type="priority" \>
</nodes>
<edges>
 <edge id="0" fromnode="0" tonode="1" priority="78" nolanes="1 speed=50.000 />
 <edge id="1" fromnode="0" tonode="2" priority="78" nolanes="1 speed = 50.000 />
 <edge id="2" fromnode="0" tonode="2" priority="78" nolanes="1 speed = 50.000 />
</edges>
```

10.7.1 Post-processing of the SUMO Street Network

After the vectorization and the generation of the street graph, the exported XML data can be post-processed for SUMO. Normally an edge means a street with only one lane and with allowed speed as 50km per hour. To obtain a realistic trace, the street graph has to be extended to bring it more toward reality. But again, not all streets have more than one lane and different kinds of streets have different number of lanes. For this purpose, a user interface is developed and integrated in the SUMO environment that provides the facility to change the attributes of a street. To get this lane information, we have used satellite images of the city. In Europe, where most of the streets indeed consist of one lane for each direction, the process is easier to be carried out by a human. Another post-processing step is to mark the road crossings that have a traffic signal installed.

10.7.2 Traffic Routes

As described above, the routes to be used by the virtual vehicles during the simulation are generated beforehand and saved as XML data. This routing information is not changeable during simulation time and must be provided along with the problem graph. A sub-program of SUMO helps in the generation of individual routes. The main input is the number of cars per time unit. A series of random routes are generated by picking the start and end nodes randomly and searching for the route between them. The generated routes are then used in SUMO for simulation.

10.8 Summary

GPS-based navigation and planning is becoming a *necessity* in modern times, where urban mobility on large scale poses a genuine challenge for sustainable transportation. Navigation systems allow users to search for a *shortest* path, based on a given metric, be it distance or travel time. However, their availability and utility is completely dependent on the underlying digital vector map.

We have seen a flexible approach to infer maps in unknown terrains using an efficient incremental clustering algorithm. Its efficiency mainly relies on two criteria, how fast a new trace can be made accessible, and how fast the cluster nodes are found. As we generate one thread for each incoming GPS trace, the approach can be used in large-scale applications where many traces are fed into the system in parallel.

Although we have described the algorithm in the context of car navigation, we highlight that the approach is fully generic and applies to most other forms of navigation on earth, e.g., hiking, cycling, etc.

All roads are annotated with travel time information for different periods of time. This allowed flexible routing queries based on the actual travel time, not just road type-based approximations, and propose variable time slots.

With the inference mechanism we provide a viewer on the accumulated map database for navigation and animation. We have adapted the approach to an on-board navigation system, so that we can expect to apply GPS routing and logging on the same GPS hand-held device. Although possible, we do not directly route on the database server itself, but export the map in form of a compressed graph that includes only crossing information. We extract the graph that is necessary and sufficient for the query.

Further tasks are to detect lanes, road types and traffic lights using data mining techniques based on the corpus of GPS data. Additionally, implementing better and self-improving filters are of both practical and academic interest.

Collaborative map generation process allows people to gather GPS traces and upload to a common web portal. The integration of these GPS traces to represent an actual road is an intensive process and poses difficult challenges to the community. Having a good base map can greatly help the integration process, allowing a good filtration of GPS traces, so that we also provided a vectorized and calibrated map extracted through raster maps.

For the vectorization and graph generation process we generated a $(0/1)$-pixel map of a topographic map by applying a variety of different digital image processing techniques. A skeletonizing algorithm then transformed this processed bitmap into a road network skeleton. In order to construct the road graph, the skeleton is made *maximally sparse*. The street graph is generated by a tracking algorithm based on the *sweep-line* paradigm that processes the skeleton by iterating on the connected components.

The seamless integration of the approach to the microscopic traffic simulation tool SUMO results in a standalone tool for urban mobility planning in the areas where vector maps are scarce.

10.9 Bibliographic Notes

There is a rich corpus on map construction and refinement for improved navigation, e.g. [80]. The work that relates closest to the approach is documented by the research lab of DaimlerChrysler [563], which also worked on map inference based on (D)GPS traces. Differently from the approach in [540], no initial map is needed in our case and hence we can infer maps for navigation even in completely unknown terrains. Another important advantage of the above approach is that the incremental construction of the map keeps the case study small, which in turn helps to make the clustering algorithm simpler and more general.

By the chosen division of the world, we additionally obtain locality that is needed to allow distributed tile processing in a large-scale application. The advantages are manifold. For example, the search for the possible road elements for the traces is reduced to a small amount, without using a geometric localization structure such as Voronoi diagrams that are more difficult to generate for data stored on disk. The separate storage of the tiles also has a large scaling potential due to a far better run-time behavior. Many traces can be integrated into an existing map in parallel. We have not refined the inference algorithm to infer lane-precise maps, which requires further statistical analyses, e.g., variants of the k-means algorithm [639]. So far, we consider only one lane for each direction. In their terms, the presented algorithm computes road centerlines. This deficiency is mainly due to the lack of more precise differential GPS information.

Several efforts have been made in the direction of extracting the road geometry from images. In [469], we see a "context-based extraction" of roads. The approach uses a set of aerial images of the same region but taken from different heights. Road extraction then proceeds iteratively by first extracting the most salient parts and the attributes of roads and then going for the finer ones. The whole process is guided by available context information. Dasen [148] discusses a somewhat similar approach for street graph extraction but does not integrate with any coordinate system or any traffic simulation. We have seen a first attempt to combine raster maps with traffic simulation for GPS trace generation.

SUMO [422] stands for *S*imulation of *U*rban *M*obility and is an open-source project developed and supported by DLR (German Aerospace Center). It was used for traffic simulation and prognosis during the world soccer championship 2006. For real-time data collection during such an enormous event, a Zeppelin airship was also used. It provided live data for the traffic simulation in SUMO that made it possible to predict the traffic jams and to inform the city authorities to redirect the traffic in advance [421].

The (improved) Douglas-Peucker algorithm goes back to [165, 337], A heuristic solution to the skeletonization problem is proposed by [57]. A comprehensive survey of skeletonizing techniques can be found in [433]. In [262], this heuristic solution has been successfully applied to find the boundaries of granulation cells.

The routing module extends the *geometric travel planning* approach as presented in [203] and [362].

Part III
Research Areas

Chapter 11
Machine Learning

In previous chapters we already looked at different machine learning problems. In Chapter 4 we introduced the concept of multi-layered and convolutional neural nets, as well as the limits and possibilities of deep learning. In Chapter 5 we have investigated randomized search algorithms for reinforcement learning, where the payoff in the rollouts is backpropagated to improve action selection. In Chapter 8 we looked at predicting time series and the problem of probabilistic inference based on background knowledge. Chapter 9 considered improvements to nearest neighbor search and classification with support vector machines for computer vision tasks. Last, but not least, in Chapter 10 we incrementally learnt a map based on incoming GPS traces.

One of the biggest driving forces to today's machine learning are advances in (parallel) computing, especially many-core systems; nowadays available not only on the central processing unit (CPU) but also on the processing unit of the graphics card (GPU). Stronger GPU systems are present, e.g., in gaming PCs or dedicated server architectures, but there is a trend of integrating GPU and CPU hardware.

While neural network often inherit a natural parallelization, in this chapter we mainly look at two state-of-the-art machine algorithms and their possible parallelization on the GPU. As a global optimizer is needed to reduce the error produced by the learning algorithms, it influences the parallelization.

We optimize an SVM, and a predictor for collaborative filtering with parallel Iterative Gradient Descent on the GPU, achieving a six-fold speedup compared to a CPU-based implementation. The reference implementations are the SVM by Leon Bottou and the BRISMF predictor from the Netflix Prices winning team.

As one crucial step we create a hash function of the input data and use it to execute threads in parallel that write on different elements of the parameter vector. We also compare the iterative optimization with a batch gradient descent and an alternating least squares optimization. The predictor is tested against over a hundred million data sets, which demonstrates the increasing memory management capabilities of modern GPUs. We make use of several techniques to alleviate the memory bottleneck.

11.1 Introduction

Roughly[1] speaking in *machine learning* we are teaching the computer to think. More sincerely, it is the systematic study of algorithms and systems that improve their knowledge or performance with experience. From a different viewpoint, machine learning has been identified as the art and science that make sense of data. A machine learning task requires a *model*, i.e., an appropriate mapping from data described by features to outputs. Obtaining such a mapping from training data is what constitutes a learning problem.

[1] This chapter is based on joint work with David Zastrau, Lorenz Hüther, Bernhard J. Berger, Stefan Edelkamp, Sebastian Eken, Lara Luhrmann, Hendrik Rothe, Marcus-Sebastian Schröder, and Karsten Sohr. It puts together and improves the work from [669, 359].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_11

The increasing relevance of graphic processors (GPUs) raises the question, to what extent parallel architectures are suitable to run arbitrary programs. Machine learning is an appropriate testbed for this question because it provides complex applications which become more and more relevant: The digital flood of information and dynamically evolving research areas such as mobile systems and robotics are only a few examples. This work develops and implements concepts to accelerate machine learning applications with a GPU-based implementation.

The applications vary between inherently parallel techniques like Neural Networks and complex probabilistic Methods like Conditional Random Fields. Also, we parallelize a SVM and a predictor for Collaborative Filtering with over one hundred million training data sets to prove the flexibility of modern GPUs.

General Purpose GPU (GPGPU) *Computing* is an ongoing field of research that has been dynamically evolving over the last few years. The continuation of Moore's Law seems to depend on the efficient application of parallel platforms. This work provides evidence that parallel programs on the GPU offer a new field of research for many machine learning algorithms.

The machine learning techniques have been chosen by the criteria of accelerated *Iterative Stochastical Gradient Descent* (ISGD) search. The main goal is to show that parallel ISGD obtains adequate precision while achieving proper speedups at the same time. We conduct two case studies.

SVMs belong to the most frequently applied machine learning approaches that can exploit ISGD in their training. SVMs are, however, not typical applications for parallelization, due to data dependencies and high memory requirements. In addition, there are very efficient CPU implementations like Leon Bottou's SVM that significantly outperform well-known libraries for the given training data, so that we take it as an appropriate benchmark for a fast sequential implementation.

Collaborative Filtering has become a relevant research subject since the public offer of the Netflix Price. The original training data set poses a challenge to the GPU memory management capabilities. Furthermore, matrix factorization is well-suited for parallel applications. We investigated if even those applications might benefit from GPGPU.

11.2 Machine Learning

Machine learning is all about using the right features to build the right models that achieve the right tasks.

Features: In essence, features define a *formal language* in which we describe the relevant objects in a domain, e.g., baked goods, their contents and production. For this case features are facets of the bakery product from the flour values (like ash content and moisture), the recipe that is used, the process parameters (like kneading time and oven temperatures), and the endproduct properties or specification. We should not normally have to go back to the domain objects, once we have found a suitable feature representation, which is why features play such an important role in machine learning.

Task: A learning task is an abstract representation of a problem we want to solve regarding the domain objects; the most common form of these is classifying them into two or more classes but there are other tasks that apply depending on the query raised. Many of these tasks can be represented as a mapping from data points to output value(s).

Model: This mapping or model is itself produced as the output of the machine learning algorithm applied to training data; there is a wide variety of models to choose from.

If data is labeled, we talk about *supervised learning*. A sample machine learning algorithm in this class is the construction of decision trees. If data is not labeled then we talk about *unsupervised learning*. A sample machine learning algorithm in this class is k-means clustering. For clustering and decision trees different storage types apply, the former storing a set of data e.g., in kd-trees and the latter using a links.

Many learning algorithms store their progress in *weights*. For SVMs these are coefficients (of the dual problem), for neural networks this is the amplifying of the edges in the network. Learning, also called training, usually takes much longer time than the application of a learned structure, e.g., for classification.

The improvement of models is known as training. There are two different circumstances that need to be considered when training a model as they have different implications for the learning process implementation. Offline learning occurs when all training instances are presented simultaneously; incremental or on-line learning occurs during the application of the trained result.

Because of the growing amount of data in companies, refined data analytics is of high importance. Large parts of the revenue model of companies like Facebook, Netflix, Walmart, or Google rely on personalized models based on the data stored from each user's actions. Often data is noisy, which means that there are statistical deviations from the true value assignments to variables. There are statistical analysis methods like the principal component analysis (PCA) to condense several features to a few important ones. Other known methods in processing data for making sense include regularization to avoid overfitting to training data with coefficients that have a high amplitude.

11.3 Neural Networks

Neural networks are highly parallel, so that the computations can often be delegated to the graphics card.

Most frequently, neural nets are used for grid-type structures like playing boards (see Chapter 4. There are, however, many cases in which data may be portrayed as a sequence of individual datapoints. Natural language, for instance, is an example of such sequential data. Here, characters form words, words form sentences and sentences form complete texts. It is undeniable that the position of the individual elements, e.g., words in a sentence, is of great importance. Since natural languages follow some explicit grammar, rearranging the words of a sentence may result in a completely different meaning or incomprehensible gibberish.

Moreover, the meaning of a single word also highly depends on its surrounding words, i.e., its context. As this example shows, not only is there information contained in the individual elements of such sequential data, but also in the structure itself. Simple feed forward neural networks, such as multilayer perceptrons, are well suited for processing static data, such as images. As described however, a lot of problems involve dynamic data, which simple feed forward architectures fail to adequately address, as they lack the means to consider temporal correlations. One popular approach that aims to solve this adds feedback loops to the network, such that the output $\hat{y}_t$ is not only computed based on the input, but a also some hidden state h_t that is being updated based on the previous state $h(t-1)$ and some input x_t.

This modular depiction is often used to describe reusable units that may occur multiple times within a single network. Thus, a neural network may be viewed as a series of modules that themselves comprise one or more layers of nodes.

Recurrent Neural Networks. Using *recurrent neural networks* (RNNs), such types of problems may be addressed. If for each time step t a prediction is made, it is possible to solve *many-to-many problems*, i.e., problems that involve finding some sequence, given another sequence. That is, since sequences may be regarded to as a series of discrete time steps as well, that may, one by one, be passed to the network as input. Then for each input step, an output can be predicted that itself forms sequence of discrete steps in time. It is also possible, however, to solve *many-to-one problems*, for example, sentiment analyses or *one-to-many problems* like generating a caption, given an given an image.

In a RNN, with each step t, the hidden state h_t is computed using a tanh-layer over the weighted previous state Wh_{t-1}, the weighted input Ux_t for matrices U,W and some bias b, thus yielding $h_t = tanh(W \cdot h_{t-1} + U \cdot x_t + b)$. Moreover, the output $\hat{y}$ at step t is given by the product of the hidden state h_t and another weight matrix V, plus some other bias c, resulting in $\hat{y}_t = V \cdot h_t + c$.

Recurrent neural networks can be very forgetful, since they use a modified version of the backpropagation technique, called *backpropagation through time*. This is necessary as with RNNs not only do the models coefficients need to be adjusted over all input samples, but also over all time steps. The loss is given by the sum of all losses L_t over all time steps $t \in T$, used for gradient calculation. This gradient also needs to be calculated for the weight matrix Wh_t that contains hidden state h_t at time step t. Unfortunately, however this involves derivatives, which tend to decrease the gradient for a larger T if Wh has eigenvalues $|\lambda|^{T-1} < 1$. This problem is often referred to as the *vanishing gradient problem*. Vice versa, in the case that $|\lambda|^{T-1} > 1$ holds true, the gradient can grow infinitely large. This is known as the *exploding gradient problem*. Both cases have the effect that RNNs cannot be trained over long distances.

LSTM Networks. As a result, RNNs suffer from a tendency to lose their effectiveness in tasks involving causalities over long sequences. *Long short-term memory* (LSTM) networks offer a solution to the *vanishing gradient problem*, but do not alleviate the effects of the *exploding gradient problem*.

LSTM networks belong to the class of recurrent neural networks and are, thus, tailored towards solving sequence prediction problems. This ability makes them well suited for complex problem domains like machine translation or speech recognition. As such they are not unlike conventional RNNs. The difference is that LSTM networks introduce the concept of cell state, denoted as c_t at time step t. For regular RNNs such module could for instance be a tanh-layer, that computes an output $\hat{y}_t$ in accordance to some input x_t and the previous hidden state h_{t-1}.

In the case of LSTM networks there are much more complicated modules involved, next to others, that make up for its architecture. Here, a single module comprises not just one, but four layers, each of which serving the purpose of manipulating the cell state c_t in accordance to the previous cell state c_{t-1}, the previous hidden state h_{t-1} and some input x_t.

These layers are often referred to as *gates* that control the flow of information through the network. First, the forget gate f, a *sigmoid*-layer, removes irrelevant information from h_{t-1} and multiplies it with c_{t-1}. Next, the update gate u, first computes a scale factor i_t through another sigmoid layer and secondly a so-called *candidate matrix* $\hat{C}_t$ using a tanh-layer, again in accordance to h_{t-1} and x_{t-1}. Finally, the product $i_t\hat{C}_t$ is added to the result of the forget operation. The last gate in the series, is the output gate o, realized by another sigmoid-layer over h_{t-1} and x_{t-1}.

Next, the *tanh* function is applied to the cell state and multiplied with the output of the sigmoid layer, yielding the output h_t. We highlight that the *tanh* operation applied to the cell state in the last step is a single operation and not an entire layer of individual nodes. It serves the purpose of ensuring that the state takes a value between -1 and 1. Moreover, it is easily visible that this architecture allows for the cell state to pass unhindered between the modules, such that it is not subject to the vanishing gradient problem.

A comprehensive list of all *gate*-operations performed within a single LSTM module is the following

$$
\begin{aligned}
i_t &= \sigma(x_t U^i + h_{t-1} \cdot W^i) \\
f_t &= \sigma(x_t U^f + h_{t-1} \cdot W^i) \\
o_t &= \sigma(x_t U^o + h_{t-1} \cdot W^o) \\
\tilde{C}_t &= tanh(x_t \cdot U^g + h_{t-1} \cdot W^g) \\
C_t &= f_t \cdot \tilde{C}_{t-1} + i_t \cdot \tilde{C}_t \\
h_t &= tanh(\tilde{C}_t) \cdot o_t
\end{aligned}
$$

Recall that W and U denote the weight matrices for the module's layers, which have been omitted in the description for better understanding of the operations.

Gated Recurrent Unit Networks. A concept similar to LSTM networks are *gated recurrent unit (GRU) networks*. Much like LSTMs, these networks also make use of gates to selectively convey information through time. In the broadest sense, GRU networks pose an enhancement to conventional LSTMs as they are easier to implement and compute. Instead of having three gates, GRU networks only possess an update gate z that decides whether to update the current hidden state h_{t-1} with the new state h_t and a reset gate r that can selectively ignore

the previous hidden state h_{t-1}. Moreover, GRU networks abandon the idea of cell state and instead rely on the hidden state only, much like the classic RNNs.

Bidirectional Sequence Models. The conventional versions of the sequence models discussed, i.e., RNN, LSTM and GRU networks all process the input in a single direction. Subsequently, such models can also only learn sequential dependencies in one single direction. Bidirectional models, such as BRNN, BLSTM or BGRU networks on the other hand can infer sequential relationships in both, forward and backward direction. Bidirectional recurrent neural networks were introduced by combining two unidirectional RNNs in a single architecture. The networks is split into two parts that are connected to the same input layer, but the input is independently processed in opposing directions. Only at the final output layer, the output of both sets of nodes is connected. This allows the network to also consider future information from the sequence, unavailable to conventional RNNs. According to Schuster and Paliwal this leads to improved predictive performance of BRNNs, compared to regular RNNs.

Graph Neural Networks. Conceptually, *graph neural networks* (GNNs) represent trainable, parametric functions over graphs. They are more involved, as they allow better correlations over distant features. As an example, aggregate-combine GNNs with L layers are specified with aggregate functions agg_i, combination functions $comb_i$ and a classification function c. On an input graph $G = (V,E)$, where V is the set of vertices and E the set of edges connecting the vertices, i.e., tuples over $V \times V$, a GNN maintains a state in form of a k-dimensional vector, x_v for each vertex v. Computation consists of updating these states throughout L iterations, with x_v^i denoting the states after iteration i. The computation model corresponds to updates

$$x_v^i = comb_i(x_v^{i-1}, agg_i x_w^{i-1}(\{\{w \in N(v)\}\})),$$

where $N(v)$ is the set of neighbors for vertex v in G, and $\{\{\ldots\}\}$ denotes a multiset (i.e., unordered set whose elements are associated with multiplicities). That is, at stage i, each vertex v receives the state of its neighbors which are then aggregated, and the result combined with the current state x_v^{i-1} to produce the next state x_v^i. The fact that agg_i maps multisets of states into real vectors means that it does not depend on the source of the received messages. GNNs are used for node or graph classification. In the first case, after the final stage, node v is classified into class $c(x_v^{(L)})$ determined by a classification function c. In the second case, the function c maps the multiset $\{\{x_v^{(L)} \mid v \in V\}\}$ into a single, scalar output; an operation referred to as a *readout*. The functions involved in the mapping from inputs to outputs can be linear or non-linear, and they are all trainable: in the supervised case by minimizing an error function defined using a training set.

11.4 GPGPU Essentials

General purpose GPU programming (GPGPU) refers to using the Graphics Processing Units (GPUs) for scientific calculations other than mere graphics. In contrast to Central Processing Units (CPUs), GPUs are programmed through kernels that are selected as threads to run on each core, which is executed as a set of threads. Each thread of the kernel executes the same code. Threads of a kernel are grouped in blocks. Each block is uniquely identified by its index and each thread is uniquely identified by the index within its block. The dimensions of the thread and the thread block are specified at the time of launching the kernel.

Programming GPUs is facilitated by APIs and supports special declarations to explicitly place variables in some of the memories (e.g., shared, global, local), predefined keywords containing the block and thread IDs, synchronization statements for cooperation between threads, runtime API for memory management (allocation, deallocation), and statements to launch functions on GPU. This minimizes the dependence of the software from the given hardware.

The memory model loosely maps to the program thread-block-kernel hierarchy. Each thread has its own on-chip registers, which are fast, and off-chip local memory, which is quite slow. Per block there is also an on-chip shared memory. Threads within a block cooperate via this memory. If more than one block is executed in parallel, then

the shared memory is equally split between them. All blocks and threads within them have access to the off-chip global memory at the speed of RAM. Global memory is mainly used for communication between the host and the kernel. Threads within a block can communicate also via light-weight synchronization.

GPUs have many cores, but the computational model is different from the one on the CPU. A core is a streaming processor with some floating point and arithmetic logic units. Together with some special function units, streaming processors are grouped together to form streaming multiprocessors. Programming a GPU requires a special compiler, which translates the code to native GPU instructions. The GPU architecture mimics a single instruction multiple data computer with the same instructions running on all processors. It supports different layers for accessing memory. GPUs forbid simultaneous writes to a memory cell but support concurrent reads.

On the GPU, memory is structured hierarchically, starting with the GPU's global memory called video RAM, or VRAM. Access to this memory is slow, but can be accelerated through coalescing, where adjacent accesses with less than word-width number bits are combined to full word-width access. Each streaming multiprocessor includes a small amount of memory called SRAM, which is shared between all streaming multiprocessors and can be accessed at the same speed as registers. Additional registers are also located in each streaming multiprocessor but not shared between streaming processors. Data has to be copied to the VRAM to be accessible by the threads.

Table 11.1: Characteristics of different APIs for parallel programming.

Properties		**API**				
		OpenMP	Pthreads	MPI	GPGPU	OpenGL
Architecture		MIMD	MIMD	MIMD	SIMD	SIMD
Synchronisation	• *lock-step*	+	+	+	+	+
	• *bulk*	+	+	+	+/-	-
	• *fine-Grain*	+	+	+	+/-	-
Model	Process-Interaction	*shared memory*	*shared memory*	*message passing*	*shared memory*	-
	• *Task Parallelism*	(+)	+	+	+	-
	• *Data Parallelism*	+	(+)	(+)	+	+
Scalability		+	-	-	+	-
Transparency		-	+	+	+	+
Overhead (implementation)		+	○	○	-	-
Overhead (resources)		+	+	○	-	-

Since frameworks like CUDA have enabled programmers to utilize the increased memory and thread management capabilities of modern GPUs there is a wider selection of applications for GPGPU (General Purpose Computation on Graphics Processing Units). Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. Thus, gradient based mini-batch or even iterative optimization techniques may be efficiently run in parallel on the GPU. Regarding flexibility and capabilities GPGPU is positioned between high-level parallel programming languages such as OpenMP and classical shader programming (see Table 11.1).

To illustrate the potential of GPGPU programming for machine learning we experimented with a Boltzman machine for solving the traveling salesman problem (TSP). Boltzmann Machines belong to the class of auto-associative networks that have one layer of neurons. They are completely connected, meaning that changes in activity of a single neuron propagate iteratively across the whole network. Boltzmann Machines do not support direct feedback, i.e., a neuron is not connected to itself. Thus, in principal auto-associative networks are no neural networks. Such a Boltzmann Machine is inherently parallel and we obtained a 487-fold speed-up for 30 towns. While the application scales almost linear on the GPU, it scales exponentially on the CPU. For more than 120 cities the memory consumption exceeds the limits of the grapics device.

We encapsulate data for both CPU and GPU and provide a unique interface. Size and indices of data fields are encapsulated, and data fields are buffered since older GPU architectures only support 32-bit words. The indices are stored in one-dimensional texture memory, since this contains a cache even in older GPU architectures and every thread frequently accesses the indices. Besides, this reduces memory complexity because data is

conglomerated in a buffer and thus data transfers are handled in one single transaction. If required, it is also possible to just copy single data fields of arbitrary size.

11.5 Iterative Gradient Descent and Parallelization

The data module is used by all applications. It encapsulates data, that exists twice on CPU and GPU and provides a unique interface. Size and indices of data fields are encapsulated and data fields are buffered since older GPU architectures only support 32-bit words (see Program 11.1).

Program 11.1: Bit stuffing on the GPU.

```
if (size % sizeof(float)!=0)
  size+=sizeof(float)-(size%sizeof(float));
```

The indices are stored in one-dimensional texture memory, since this contains a cache even in older GPU architectures and every thread frequently accesses the indices. Besides, this reduces memory complexity because data is conglomerated in a buffer and thus data transfers are handled in one single transaction. If required it is also possible to just copy single data fields of arbitrary size.

Iterative Stochastic Gradient Descent (ISGD) approximates the true gradient for each new training example by $\theta = \theta - \eta \sum_{i=1}^{N} \nabla L(\theta_i)$, where θ is a weight vector, η is the (adaptive) learning rate and L is some loss function. ISGD is inherently sequential and tends to converge to local minima for non-convex problems. As a compromise θ may be updated by minibatches, consisting of the sum of several training examples. The idea of minibatches complements the semi-parallel CUDA programming paradigm. ISGD converges to a good global solution, while the parallel computation of the gradients is likely to produce poor results because the parallel processing of the input data has the negative side effect that threads do not profit from and even more do not consider the changes in the objective that other threads are performing at the same time. A hybrid approach is to use the non-optimal parallel solution to rapidly converge to some adequate solution and then further improve this solution by using the CPU-based solution. This approach combines the shorter execution time for one training iteration on the GPU with the better precision on the CPU.

Note that the time for data transfer alone often exceeds the complete CPU-based training time. Therefore, it is necessary to also implement the validation on the GPU. Although the validation only requires reading access, we can also adopt the memory access pattern from the training procedure here.

Bottou states ISGD is well suited for SVMs because the problem is based on a simple convex objective function. It also applies well to Collaborative Filtering. Even for SVMs we found that almost 70% of the CPU instructions are used for vector addition and scalar products, an indicator that the application might benefit from GPGPU. But since the vector length is most often limited to a few dozen elements, standard functions such as those from the CUBLAS library are practically not applicable. The input data is already provided as support vectors, which are used to fix θ in each episode. Since the vector lengths vary greatly, they cannot be simply partitioned on thread blocks with a fixed number of threads. Additionally, each training episode requires numerous memory accesses to θ that do not exhibit spatial locality which could be efficiently exploited by the VRAM-controller. As a solution to this problem θ might be loaded into shared memory. Considering the limited shared memory size of only 64 KB the training data has to be loaded piecewise and a hash function has to be defined so that every thread may infer its input data from its thread ID (see Figure 2.4). In other words the hash function allows a block of threads to load exactly those elements of θ into shared memory which are needed for the training data that has been assigned to this block.

For Collaborative Filtering we found that over 40% of the CPU instructions are used for the computation of the probabilities of the subsequent states and the maximization of the Log-Likelihood. The remainder of the instructions is mainly used for memory management. This is caused by memory allocation for numerous vector and matrix operations. Therefore, the update needs to be implemented in a way that conditional probabilities are loaded into shared memory at least to some extent.

11.6 Applications

Collaborative Filtering Matrix Factorization is based on the idea that any matrix $\mathbb{R} \in \mathbb{R}^{N \times M}$ with ratings can be approximated by a matrix $P \in \mathbb{R}^{N \times K}$ of user IDs and a matrix $Q \in \mathbb{R}^{K \times M}$ with the article's IDs: $R \approx PQ$. Here N is the number of users, M is the number of articles and K is the number of parameters, that are used to characterize those. The bigger one chooses K, the more precisely R can be approximated. This approach holds the advantage to generalize to non-existent ratings based on two low-dimensional matrices. The ISGD update in each step for user p_{uk} and article q_{ki} is:

$$p'_{uk} = p_{uk} + \eta^p(u,i,k) \cdot (e_{ui} \cdot q_{ki} - \lambda^p(u,i,k) \cdot p_{uk})$$
$$q'_{ki} = q_{ki} + \eta^q(u,i,k) \cdot (e_{ui} \cdot p_{uk} - \lambda^q(u,i,k) \cdot q_{ki}).$$

To compare the ISGD to a batch optimization we also implemented an Alternating Least Squares optimization on the GPU where the update step is basically $\mathbf{p}_u = \mathbf{W}_u \mathbf{d}_u$, where d_u denotes the input-output covariance vector and W_u is the updated inverted covariance matrix of input. This technique is also based on matrix factorization but yields the advantage that P and Q are alternately being updated so that either P or Q can be treated as in- or output and be written in parallel. The learning refers to the the methods in Algorithm 11.1.

In the algorithm P and Q are being compressed to the required dimensions. Afterwards a hash function ϕ is being created that maps every user to the movies he has rated. Thus, multiple threads can simultaneously process the ratings by one user in a shared memory. Next the training data is transferred to the GPU and the optimization is being performed. Note that batch (ALS) and mini-batch (ASGD) optimization need an extra step to load the data into the shared memory.

As application we choose the Netflix competition. First, Netflix provided with over hundred million user ratings the biggest real data set for collaborative filtering so far. Secondly, during the competition many interesting machine learning techniques have been developed. Two of them, both based on matrix factorization, will be accelerated by the GPU in this work. Netflix uses an AI-based system to recommend movies to users based on their previous purchases. The system that Netflix used until the conclusion of the competition had a root mean squared error (RMSE) of 0.95256.

Töscher et al., who won the competition with a final root mean squared error of 0.8554, used (amongst others) an estimator called *Biased Regularized Incremental Simultaneous Matrix Factorization* (BRISMF). It has been introduced in the context of a progress report for the Netflix competition. It also uses SGD.

Figure 11.1 shows the profile of time vs. accuracy for the implementation of BRISM using the Netflix data. We see that the naive parallelization gives good results. The error (0.9101) is slightly bigger than the original one (0.9068) on the other hand we measure a 1088/180 = 6.04 speed-up. It should be noted that the overall precision for both programs increases if we increase K. A comparison between SGD, ASGD and ALS showed that alternating SGD yields the worst results. Although ASGD gave a $8.1/2.7 = 3$-fold speed-up and always the same results, it converged to 0.941, as opposed to 0.922 for SGD on the CPU (for $K = 10$). While a greater value for K gave up to 9-fold speedup the precision remained on a clearly lower level.

SVM Raw data presented to a supervised statistical machine learning algorithm is often mapped to a set of numerical values, called the *feature vector*. The classification problem deals with the prediction of the labels l of previously unknown feature vectors $\mathbf{x} \in \mathbb{R}^d$ that constitute the test data. During training, a partitioning of

Algorithm 11.1: Some basic machine learning algorithms.

Algorithm A (Boltzmann Oscillation)
Choose sufficiently large temperature parameter T
repeat
  repeat
    Choose random neuron k
    Calculate new output value o_k
  until Average energy level remains constant
  Decrease temperature (e.g. $T = 1/(\text{number of time steps})$)
until steady state

Algorithm B (Training algorithm for BRISMF)
for $(u,i) \in \mathscr{R}$ **do**
  calculate e_{ui}
  for $k \in \{1,\ldots,K\}$ **do**
    update p_u, the u-th row of P, and q_i, the i-th column of Q

Algorithm C (Recalculating p_u)
$B_u \leftarrow \lambda n_u I$
$W_u \leftarrow 0 \cdot I$, if $\lambda = 0$
$W_u \leftarrow 1/(\lambda \cdot n_u) \cdot I$, otherwise
$d_u \leftarrow 0$
for $i : (u,i) \in \mathscr{R}$ **do**
  $B_u \leftarrow B_u + q_i \cdot q_i^T$
  $W_u \leftarrow W_u - (W_u q_i \otimes q_i^T W_u)/(1 + q_i^T W_u q_i)$
$d_u \leftarrow d_u + q_i - r_{ui}$
$p_u \leftarrow W_u d_u$

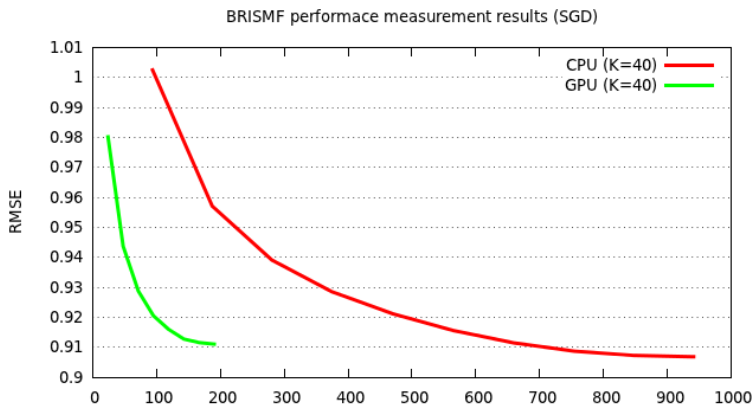


Figure 11.1: BRISMF time-accuracy trade-off for $k = 40$.

the feature space $\mathbb{R}^d$ is learned, where each partition is assigned a label based on a set of training samples with known label. The challenge is to approximate the unknown distribution without overfitting the training data. SVMs achieve this task by learning coefficients for a kernel mapping to a high-dimensional space, where a linear class border is spanned up by several support vectors that outline the data. Theoretically, it is sufficient to determine the class border by just three support vectors. However, it is not known in advance if any of the known kernels realizes a suitable mapping. The use of generic kernels instead leads to a much larger number of support vectors (which critically influence classification time). In the worst case finding a separating hyperplane takes quadratic time in the number of data points.

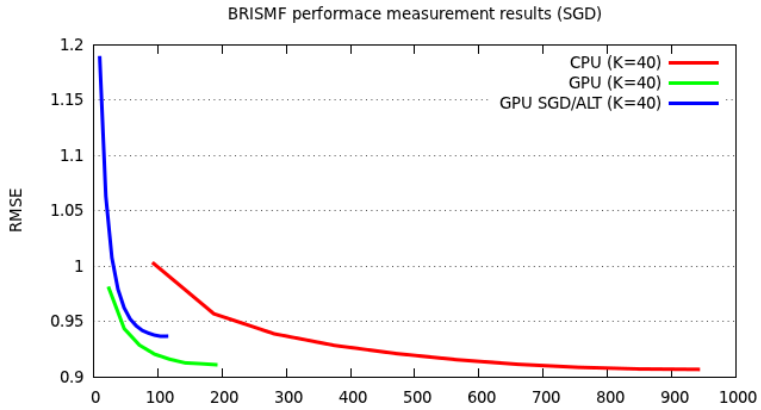


Figure 11.2: SGD and ASGD for $K = 40$.

Leon Bottou used a SVM to classify text documents. He applied stochastic gradient descent for training and classifying wrt a linear SVM. This already gives good results after a very short time (order-of-magnitudes speed-up) compared to other libraries.

The compression of the input vectors is implemented with an STL-vector and has been accelerated with OpenMP. To generate the hash function θ efficiently, an integer array containing sufficiently many addresses is precomputed (see Program 11.2).

Program 11.2: Generating θ.

```
int n = 0;
for (id = min; id < max ;++id) {
  weight = input[id] ;
  if (hash[weight].map >= theta.size)
    hash [weight].map = n++;
  output[id] = hash[weight].map;
}
```

To speed up the data transfer floating point numbers are compressed to 16-bit integers on the CPU (via the OpenEXR-library) and extracted on the GPU (via `half2float`), which is very accurate especially for input values near zero and does not affect the overall precision. Threads collaborate block-wise during training. At first all weights, which are required by the threads, are loaded into shared memory. Then comes a thread barrier. Finally, each thread processes and adds the delta to the shared memory. The mapping Shared Memory into Global Memory is implemented in the hash function. Afterwards there is another thread barrier before the threads collaboratively write the delta from the shared memory to the global memory, i.e. add it to θ. Loading the data works analogously for the validation. Each thread checks for the correct classification and adds 1 to the global error counter in case it's wrong.

The training data size is substantial ($\approx$ 350 Megabyte). Since training takes only about 1.4 seconds and the training data must be uploaded to the GPU, the best possible speed-up was limited by a factor of about 2.

11.7 Summary

In this chapter we showed that GPUs are suited to accelerate other machine learning applications aside from image processing and matrix operations. We used different optimization techniques to minimize the memory requirements on the GPU and were able to process hundreds of megabytes on the GPU efficiently.

Parameters have to be adjusted to the specific GPU architecture. We tested local as well as global gradients and compared speed, precision and scalability of each method. While we were able to achieve a huge speed-up with the Boltzmann Machine, we were able to accelerate Collaborative Filtering by a factor of 6. The GPU version of the SVM only showed a 1.66-fold speed-up but according to a highly tuned reference impelmetation.

In additional experiments for the Boltzmann machine we achieved a 487 times speed-up with the Boltzmann Machine, and the Conditional Random Field was about three times faster. The success of deep learning with convolutional neural nets also relies on fast parallel learning on the GPU.

As GPU computations have a bad reputation in terms of energy usage we measured the power consumption for GPU and CPU, which becomes relevant for mobile devices. Abbreviating P for power (in Watts) and C for Current (in VA), we see that due to the acceleration the overhead is at most a factor of 2.

	CPU		GPU	
Technique	P (W)	C (VA)	P (W)	C (VA)
BRISMF	156–158	166–168	204–207	210–215
SVM	119–123	134–136	241–252	251–262

11.8 Bibliographic Notes

Machine learning [258] is probably the most successful field in AI. Learning approaches include *Concept Learning:* e.g., Version Space and Candidate Elimination; *Knowledge Representation and Reosoning:* e.g., Bayes', Neural, and Graphical Nets; *Classification:* e.g., Decision Trees, Neuronal Nets and Perceptrons, SVMs; *Clustering and Regression:* e.g. based on partitioning (k-means, k-medoid), densities (dbscan) and hierarchies (single-link, cure, optics); *Rule Learning:* Concepts, Words, Macros, Association Rules; *Reinforcement Learning:* Value and Policy Iteration, Real-Time Dynamic Programming, UCT and NRPA; *Recommender Systems:* Collaborative Filtering; Matrix Factorization, Latent Features; *Regular Languages and Strings:* Finite-State Automata Learning, SAX; *Evolutionary Learning:* Genetic Algorithms and Particle Swarm Algorithms.

The Netflix price is presented by [19]. Kato and Hosino [385] claim that they were able to speed up the training for Singular Value Composition [647] by a factor of 20. In this work they use the same gradient as [647] as well as their own algorithm for matrix compression. However, they just use randomly generated data, and they do not give information regarding the results precision.

Support vector machines go back to [141, 561, 630]. Bottou's implementation [72] is significantly faster than LIBSVM (but can deal only with linear kernels). Catanzaro et al. [104] used GPGPU to achieve a nine to 35-times speed-up compared to training with LIBSVM Classification was even 81 to 138-times faster.

Both implementations used *Sequential Minimal Optimization* [516]. However, they didn't implement regression and no 32-bit floating-point arithmetic. The software package by Carpenter [102] also uses Sequential Minimal Optimization to optimize SVMs and supports regression as well as 64-bit floating point arithmetic. Their code runs 13 to 73 times faster for training and 22 to 172 faster for classification than the CPU reference implementation.

Preliminary results [3] reduced the gap between CPU and GPU performance in deep learning applications. In the area of reinforcement learning and in the context of asynchronous algorithms, CPU-only algorithms already

achieve a competitive performance. Extended tuning of performance of asynchronous reinforcement learning algorithms on large computer clusters brought the training time down from hours to minutes.

There are many machine learning libraries, such as Weka: about 20 years of development; written in Java, covers many important learning algorithms; RapidMiner: predictive analytics; LibSVM: mainly methods for learning with SVMs; Torch: Multi-dimensional array (tensor) handling, where the core programming language is LUA, supports deep learning and stochastic gradient descent; and TensorFlow: combining different learning schemes.

RNN were discussed by [115]. LSTM were first introduced by Hochreiter and Schmidhuber [341]. Schuster and Paliwal [566] introduced bidirectional recurrent neural networks. Graph neural networks were used to represent trainable, parametric functions over graphs by [317].

Chapter 12
Problem Solving

This work will merge two different lines of research, namely state-space search with binary decision diagrams (BDDs), that was initially proposed for Model Checking and still is state of the art in AI Planning; and state-space compaction with (minimal) perfect hashing, which is used in the algorithm community as a memory-based index for big data (often residing on disk).

We will see how BDDs can serve as the internal representation of a perfect hash function with linear-time ranking and unranking, and how it can be used as a static dictionary and an alternative to the recent compression schemes exploiting hypergraph theory. This will also result in a simple method to split a BDD in parts of equal number of satisfying assignments and to generate random inputs for any function represented as a BDD. As a surplus, the BDD-based hash function is monotone.

In terms of applications, symbolic exploration with BDD constructs a succinct representation of the state space. For each layer of the search, a BDDs is generated and stored, and will later serve as an index to do extra work like the classification of game states. Based on this approach we will study, how to strongly solve a game in a combination of symbolic and explicit-state space exploration.

12.1 Introduction

Hashing[1] is an efficient methods for solving combinatorial problems by mapping each state to a location in memory. To avoid collisions, perfect hash functions serve as compressed representations of the search space and support the execution of exhaustive search algorithms like breadth-first search and retrograde analysis.

Perfect hashing computes the *rank* of a state, while the inverse operation *unrank* reconstructs the state given its rank. Efficient bitvector algorithms are derived and generalized to a larger variety of games. We study *rank* and *unrank* functions for permutation games with distinguishable pieces, for selection games with indistinguishable pieces, and for general reachability sets. The running time for ranking and unranking in all three cases is linear in the size of the state vector.

To overcome space and time limitations in solving games like *Frogs-and-Toads* and *Fox-and-Geese*, we utilize parallel computing power in form of multiple cores on modern central processing units (CPUs) and graphics processing units (GPUs). We obtain an almost linear speedup with the number of CPU cores. Due to the much larger number of cores, even better speed-ups are achieved on GPUs.

We also combine bitvector and symbolic search with BDDs that compactly represent state sets. The hybrid algorithm for strongly solving general games initiates a BDD-based solving algorithm, which consists of a

[1] This chapter is based on joint work with Damian Sulewski, Cengizhan Yücel, Peter Kissmann, Martin Dietzfelbinger and Martha Rohte. It puts together and improves the work from [228, 210, 209, 159].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_12

forward computation of the reachable state set, possibly followed by a layered backward retrograde analysis. If the main memory becomes exhausted, it switches to explicit-state two-bit retrograde search. We take *Connect Four* as a case study.

Strong computer players for combinatorial games like *Chess* or *Go* have shown the impact of advanced search engines. For many games they play on expert level, sometimes even better. For some games like *Checkers* the solvability status of the initial state has been computed: the game is a draw, assuming optimal play of both players.

We consider *strongly solving* a game in the sense of creating an optimal player that returns the best move for *every* possible state. After computing the game-theoretical *value* of each state, the best possible action is selected by looking at the values of all successor states. In many single-agent games the value of a game simply is its goal distance, while for two-player games the value is the best possible reward assuming that both players play optimally.

We apply *perfect hashing*, where a perfect hash function is a one-to-one mapping from the set of states to some set $\{0,\ldots,m-1\}$ for a sufficiently small number m. *Ranking* maps a state to a number, while *unranking* reconstructs a state given its rank. One application of ranking (and unranking) functions is to compress (and decompress) a state.

We will see that for many games, space-efficient perfect hash functions can be constructed prior to the search. In some cases, it is even possible to devise a family of perfect hash functions, one for each (forward or backward) search layer. We introduce linear time algorithms for invertible perfect hashing for

- *permutation games*, i.e., games with distinguishable pieces. In this class we find *Sliding-Tile* puzzles with numbered tiles, as well as *Top-Spin* and *Pancake* problems. The *parity* of a permutation will allow us to restrict the range of the hash function. There are other games like *Blocksworld* that belong to this group.
- *selection games*, i.e., games with indistinguishable objects. In this class we find tile games like *Frogs-and-Toads*, as well as strategic games like *Peg-Solitaire* and *Fox-and-Geese*. There are other games like *Awari*, *Dots-and-Boxes*, and *Nine-Men's-Morris*, that can be mapped to this group.

For analyzing the state space, we utilize a bitvector that covers the solvability information of all reachable states. Moreover, we apply symmetries to reduce the time- and space-efficiencies of the algorithms. Besides the design of efficient perfect hash functions that apply to a wide selection of games, we compute successor states on multiple cores on the central processing unit (located on the motherboard) and on the graphics processing unit (located on the graphics card).

For general state spaces, we look at explicit-state and symbolic hashing options that apply once the state space is generated. As an example, perfect hashing with binary decision diagrams (BDDs) is applied to strongly solve *Connect Four*.

What is a BDD? A BDD is a directed acyclic graph data structure for a Boolean function. Nodes are labeled with variables, edges and sinks are labeled with 1 and 0. Redundant nodes are eliminated and the variable ordering is the same on every path.

What do BDDs represent? BDDs are characteristic functions of planning state sets. Each path from root to the 1-sink acts as the binary representation of one state in the set. Initial and goal conditions can also be represented as such state sets.

How does search with BDDs work? Actions are cast as a set representation of their pre- and postconditions (transition relation). In the *image* symbolic exploration checks the pre- and applies the postcondition.

What is the advantage of BDDs? Firstly, BDDs can have an advantage in space. Many polynomial-sized BDDs represent exponentially many states. Secondly, forward and backward exploration are the same, except that the initial and goal condition as well as the pre- and postcondition are exchanged. Thirdly, the compact representation of many states in a smaller structure often results in faster runtimes.

12.2 Perfect Hashing

A *hash function h* is a mapping of a universe U to an index set $\{0,\ldots,m-1\}$. The set of reachable states S of a search problem is a subset of U, i.e., $S \subseteq U$. We are interested in injective hash functions, where a mapping is injective, if for all $f(x) = f(y)$ we have $x = y$. A hash function $h : S \to \{0,\ldots,m-1\}$ is *perfect* if for all $s \in S$ with $h(s) = h(s')$ we have $s = s'$. The *space efficiency* of a hash function $h : S \to \{0,\ldots,m-1\}$ is the proportion $m/|S|$ of available hash values to states.

Given that every state can be viewed as a bitvector and interpreted as a number, one inefficient design of a perfect hash function is immediate. The space requirements of the corresponding hash table are usually too large. A space-optimal perfect hash function is bijective. A perfect hash function is *minimal* if its space efficiency is 1, i.e., if $m = |S|$.

Efficient and minimal perfect hash functions allow direct-addressing a bitstate hash table instead of mapping states to an open-addressed or chained hash table. The computed index of the direct access table uniquely identifies the state.

Whenever the average number of required bits per state for a perfect hash function is smaller than the number of bits in the state encoding, an implicit representation of the search space is fortunate, assuming that no other tricks like orthogonal hashing apply.

Two hash functions h_1 and h_2 are *orthogonal* if for all states s, s' with $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$ we have $s = s'$. In case of orthogonal hash functions h_1 and h_2, the value of h_1 can, e.g., be encoded in the file name, leading to a partitioned layout of the search space, and a smaller hash value h_2 to be stored explicitly. If the two hash functions $h_1 : S \to \{0,\ldots,m_1-1\}$ and $h_2 : S \to \{0,\ldots,m_2-1\}$ are orthogonal, their concatenation (h_1,h_2) is perfect: for two hash functions h_1 and h_2 and any state s in S with $(h_1(s),h_2(s)) = (h'_1(s),h'_2(s))$ we have $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$. Since h_1 and h_2 are orthogonal, this implies $s_1 = s_2$.

The other important property of a perfect hash function for a state space search is that the state vector can be reconstructed given the hash value. A perfect hash function h is *inversible* if given $h(s)$, $s \in S$ can be reconstructed. The inverse h^{-1} of h is a mapping from $\{0,\ldots,m-1\}$ to S. Computing the hash value is denoted as *ranking*, while reconstructing a state given its rank is denoted as *unranking*.

For the exploration of the search space, in which array indices serve as state descriptors, inversible hash functions are required. For the design of minimal perfect hash functions in permutation games, *parity* will be a helpful concept. An *inversion* in a permutation $\pi = (\pi_1,\ldots,\pi_n)$ is a pair (i,j) with $1 \le i < j \le n$ and $\pi_i > \pi_j$. The *parity* of the permutation π is defined as the parity (*mod* 2 value) of the number of inversions in π, A permutation game is *parity-preserving* if no move changes the parity of the permutation. Parity-preservation allows us to separate soluble from insolvable states in several permutation games. If the parity is preserved, the state space can be compressed.

A property $p : S \to \mathbb{N}$ is *move-alternating*, if the parity of p toggles for every action, i.e., for all s and $s' \in succs(s)$ we have $p(s')\ mod\ 2 = (p(s)+1)\ mod\ 2$. As a result, $p(s)$ is the same for all states s in one BFS layer. States s' in the next BFS layer can be separated by knowing $p(s') \neq p(s)$. One example for a move-alternation property is the position of the blank in the sliding-tile puzzle.

12.3 Bitvector State Space Search

In *two-bit breadth-first search* every state is expanded at most once. The two bits encode values in $\{0,\ldots,3\}$ with value 3 representing an unvisited state, and values 0, 1, or 2 denoting the current search depth *mod* 3. This allows us to distinguish generated and visited states from ones expanded in the current breadth-first layer. Note that in some cases it is possible to generate the entire state space using one bit per state. As such search does not distinguish between states to be expanded next (*open* states) and states already expanded (*closed* states),

such a *one-bit reachability* algorithm determines all reachable states but may expand a state more than once. Additional information extracted from a state can improve the running time by decreasing the number of states to be reconsidered (*reopened*).

For some domains, one bit per state suffices for performing breadth-first search. In *Peg-Solitaire* the number of remaining pegs uniquely determine the breadth-first search layer, so that one bit per state can separate newly generated states from expanded ones. This halves the space needed compared to the more general two-bit breadth-first search routine. In the event of a *move-alternation property* we, therefore, can perform breadth-first search using only one bit per state. One important observation is that not all visited states that appear in previous BFS layers are removed from the current search layer.

We next consider *two-bit retrograde analysis*. Retrograde analysis classifies the entire set of positions in backward direction, starting from won and lost terminal ones. Moreover, partially completed retrograde analyses have been used in conjunction with forward-chaining game playing programs to serve as endgame databases. Retrograde analysis works well for all games, where the game positions can be divided into different layers, and the layers are ordered in such a way that movements are only possible in between a layer or from a higher layer to a lower one. Then, it is sufficient to do the lookup in the lower layers only once during the computation of each layer. Thus, the bitstate retrograde algorithm is divided into three stages: during initialization all positions that are won for one player are marked. Then, the successors are searched in the lower layers, and, then, an iteration over the remaining unclassified positions. As a result, it is sufficient to consider only successors in the same file.

In the second part a position is marked as won if it has a successor that is won for the player to move, otherwise the position remains unsolved. Even if all successors in the lower layer are lost for one position, then this position remains unsolved. A position is only marked as lost in the third part of the algorithm, because only then it is known what all the successors are. If there are no successors in the third part, then the position is marked as lost.

Provided additional state information indicating the player to move, bitstate retrograde analysis for zero-sum games requires two bits to denote if a state is *unsolved*, a *draw*, *won for the first player*, or *won for the second player*.

Bitstate retrograde analysis applies backward BFS starting from the states that are already decided. For the sake of simplicity, in the implementation we first look at two-player zero-sum games that have no draw. Based on the players' turn, the state space is in fact twice as large as the mere number of possible game positions. The bits for the first player and the second player to move are interleaved, so that the turn can be computed by looking at the *mod 2* value of a state's rank.

12.4 Hashing Permutation Games

The *lexicographic rank* of a permutation is the position in the lexicographic order of its state vector representation. In the lexicographic ordering of a permutation $\pi = (\pi_0, \ldots, \pi_{n-1})$ of $\{0, \ldots, n-1\}$ we first have $(n!-1)$ permutations that begin with 0, followed by $(n!-1)$ permutations that begin with 1, etc. This leads to the following recursive formula: $\textit{lex-rank}((0),1) = 0$ and $\textit{lex-rank}(\pi, n) \leq \pi_0 \cdot (n-1)! + \textit{lex-rank}(\pi', n-1)$, where $\pi'_i = \pi_{i+1}$ if $\pi'_i > \pi_0$ and $\pi'_i = \pi_i$ if $\pi'_i < \pi_0$.

The lexicographic rank of permutation π (of size n) is determined as $\text{lex-rank}(\pi, n) = \sum_{i=0}^{N-1} d_i \cdot (N-1-i)!$ where the vector d of coefficients d_i is called the *inverted index* or *factorial base*. The coefficients d_i are uniquely determined. The parity of a permutation is known to match $(\sum_{i=0}^{N-1} d_i)$ *mod* 2. In the recursive definition of *lex-rank* the derivation of π' from π makes an according ranking algorithm non-linear.

Given that many existing ranking and unranking algorithms with respect to the lexicographic ordering are slow, we study a more efficient ordering based on the observation that every permutation can be generated uniformly

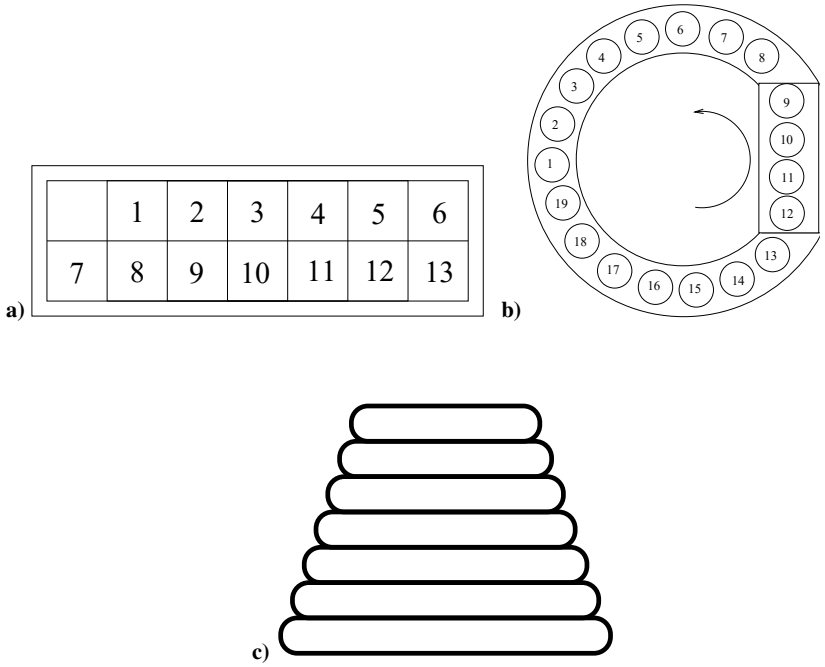


Figure 12.1: Permutation Games: a) Sliding-Tile Puzzle, b) Top-Spin Puzzle, c) Pancake Problem.

by swapping an element at position i with a randomly selected element $j > i$, while i continuously increases. The sequence of j's can be seen as the equivalent to the factorial base for the lexicographic rank.

We show that the parity of a permuation can be derived on the fly in the unranking algorithm proposed by Myrvold and Ruskey (see Chapter 1). The input is the number of elements N to permute, the permutation π, and its inverse permutation π^{-1}. The output is the rank of π. As a side effect, we have that both π and π^{-1} are modified. Fortunately, the parity of a permutation for a rank r in Myrvold and Ruskey's permutation ordering can be computed on-the fly with the unrank function.

Sliding-Tile Puzzle

Next, we consider permutation games, especially the ones shown in Figure 12.1. The $(n \times m)$ *sliding-tile puzzle* consists of $nm-1$ numbered tiles and one empty position, called the blank. In many cases, the tiles are squarely arranged, such that $m = n$. The task is to re-arrange the tiles such that a certain terminal tile arrangement is reached. Swapping two tiles toggles the permutation parity and, in turn, the solvability status of the game. Thus, only half the $nm!$ states are reachable.

We observe that, in a lexicographic ordering, every two adjacent permutations with lexicographic rank $2i$ and $2i+1$ have a different solvability status. In order to hash a sliding-tile puzzle state to $\{0,\ldots,(nm)!/2-1\}$, we can, therefore, compute the lexicographic rank and divide it by 2. Unranking is slightly more complex, as it has to determine which of the two permutations π_{2i} and π_{2i+1} of the puzzle with index i is reachable.

There is one subtle problem with the blank. Simply taking the parity of the entire board does not suffice to compute a minimal perfect hash value in $\{0,\ldots,nm!/2\}$, as swapping a tile with the blank is a move that does not change the parity. A solution to this problem is to partition the state space with respect to the position of the blank, since for exploring the $(n \times m)$ puzzle it is equivalent to enumerate all $(nm-1)!/2$ orderings together with the nm positions of the blank. If $S_0,\ldots,S_{nm-1}$ denote the set of "blank-projected" partitions, then each set S_j, $j \in \{0,\ldots,nm-1\}$, contains $(nm-1)!/2$ states. Given the index i as the permutation rank and j it is simple to reconstruct the puzzle's state.

As a side effect of this partitioning, horizontal moves of the blank do not change the state vector, thus the rank remains the same. Tiles remain in the same order, preserving the rank. Since the parity does not change in this puzzle we need another move alternating property, and find it in the position of the blank. The partition into buckets $S_0,\ldots,S_{nm-1}$ has the additional advantage that we can determine whether the state belongs to an odd or even layer.

For such a factored representation of the sliding-tile puzzles, a refined exploration retains the breadth-first order, by means that a bit for a node is set for the first time in its BFS layer. The bitvector *Open* is partitioned into *nm* parts, which are expanded depending on the breadth-first *level*.

As mentioned above, the rank of a permutation does not change by a horizontal move of the blank. This is exploited by writing the ranks directly to the destination bucket using a bitwise-or on the bitvector from layer $level-2$ and *level*. The vertical moves are unranked, moved and ranked. When a bucket is done, the next one is skipped and the next but one is expanded. The algorithm stops when no new successor is found.

Top-Spin Puzzle

The next example is the (n,k)-Top-Spin Puzzle, which has n tokens in a ring. In one twist action k consecutive tokens are reversed and in one slide action pieces are shifted around. There are $n!$ different possible ways to permute the tokens into the locations. However, since the puzzle is cyclic only the order of the different tokens matters and thus there are only $(n-1)!$ different states in practice. After each of the n possible actions, we thus *normalize* the permutation by cyclically shifting the array until token 1 occupies the first position in the array.

For an even value of k (the default) and odd value of $n > k+1$, the (normalized) (n,k) Top-Spin Puzzle has $(n-1)!/2$ reachable states. As the parity is even for a move in the (normalized) (n,k) Top-Spin Puzzle for an odd value of $n > k+1$, we obtain the entire set of $(n-1)!$ reachable states.

Pancake Problem

The n-Pancake Problem is to determine the number of flips of the first k pancakes (with varying $k \in \{1,\ldots,n\}$) necessary to put them into ascending order. No other than Bill Gates showed that $(5n+5)/3$ flips always suffice, which was improved to $18n/11$. One needs at least $15n/14$ flips. In the *n-Burned-Pancake* variant, the pancakes are burned on one side and the additional requirement is to bring all burned sides down. For this version it is known that $2n-2$ flips always suffice and that $3n/2$ flips are necessary. Both problems have n possible operators. The pancake problem has $n!$ reachable states, the burned one has $n!2^n$ reachable states. For an even value of $\lceil (k-1)/2 \rceil$, $k > 1$, the parity changes, while for an odd one, the parity remains the same.

12.5 Hashing Selection Games

Figure 12.2: Initial States in *Fox-and-Geese*.

Fox-and-Geese is a two-player zero-sum game. The lone fox (F) attempts to capture the geese (G), while the geese try to hem the Fox, so that he can't move. It is played upon a cross-shaped board consisting of a 3×3 square of intersections in the middle with four 2×3 areas adjacent to each face of the central square. One board with the initial layout is shown in Figure 12.2. Pieces can move to any empty intersection around them (also diagonally). The fox can additionally jump over a goose to capture it. Geese cannot jump. The geese win if they surround the fox so that it cannot move. The fox wins if it captures enough geese that the remaining geese cannot surround him. *Fox-and-Geese* belongs to the set of *asymmetric* strategy games played on a cross-shaped board. The chances for 13 geese are assumed to be an advantage for the fox, while for 17 geese the chances are assumed to be roughly equal.

The game requires a strategic plan and tactical skills in certain battle situations. The portions of tactic and strategy are not equal for both players, such that a novice often plays better with the fox than with the geese. A good fox detects weaknesses in the set of geese (unprotected ones, empty vertices, which are central to the area around) and moves actively towards them. Potential decoys, which try to lure the fox out of his burrow must be captured early enough. The geese must work together and find a compromise between risk and safety. In the beginning it is recommended to choose safe moves, while toward the end of the game it is recommended to challenge the fox to move out to fill blocked vertices.

```
O O O           X X X
O O O           X X X
O O   X X  ->   X X   O O
    X X X           O O O
    X X X           O O O
```

Figure 12.3: Initial and goal state in *Fore and Aft*.

```
    X X X             O O O
    X X X             O O O
X X X X X X X     O O O O O O O
X X X O X X X  ->  O O O X O O O
X X X X X X X     O O O O O O O
    X X X             O O O
    X X X             O O O
```

Figure 12.4: Initial and goal state in *Peg Solitaire*.

The *Fore and Aft* puzzle (see Figure 12.3) has been made popular by the American puzzle creator Sam Loyd. It is played on a part of the 5×5 board consisting of two 3×3 subarrays at diagonally opposite corners. They overlap in the central square. One square has eight black pieces and the other has eight white pieces, with the center left vacant. The objective is to reverse the positions of pieces in the lowest number of moves. Pieces can slide or jump over other pieces of any color. *Frogs-and-Toads* generalizes *Fore and Aft* and large boards are yet unsolved.

In *Peg-Solitaire* (see Figure 12.4) the set of pegs is iteratively reduced by jumps. The problem can be generalized to an arbitrary graph with n holes. As the number of pegs denotes the progress in playing *Peg-Solitaire*, we may aim at representing all boards with k of the $n-1$ possible pegs, where n is the number of holes. In fact, the breadth-first level k contains at most $\binom{n}{k}$ states. In contrast to permutation games, pegs are indistinguishable, and call for a different design of a hash function and its inverse.

Such an invertible perfect hash function of all states that have $k = 1, \ldots, n$ pegs remaining on the board reduces the RAM requirements for analyzing the game. As successor generation is fast, we will need an efficient hash function (rank) that maps bitvectors $(s_0, \ldots, s_{n-1}) \in \{0,1\}^n$ with k ones to $\{0, \ldots, \binom{n}{k} - 1\}$ and back (unrank). There is a trivial ranking algorithm that uses a counter to determine the number of bitvectors passed in their

lexicographic ordering that have k ones. It uses linear space, but the time complexity by traversing the entire set of bitvectors is exponential. The unranking algorithm works similarly, with matching exponential time performance.

The design of a linear time ranking and unranking algorithm is not obvious. The pieces on the board are not labeled, their relative ordering does not matter.

Hashing with Binomial Coefficients

An efficient solution for perfect and invertible hashing of all bitvectors with k ones to $\{0,\ldots,\binom{n}{k}-1\}$ utilizes binomial coefficients that can either be precomputed or determined on the fly. The algorithms rely on the observation that once a bit at position i in a bitvector with n bits and with j zeros is processed, the binomial coefficient $\binom{i}{j-1}$ can be added to the rank. The notation $\max\left\{0,\binom{i}{zeros-1}\right\}$ is shorthand notation to say, if $zeros < 1$ take 0, otherwise take $\binom{i}{zeros-1}$.

The time complexities of both algorithms are $O(n)$. In case the number of 0s exceeds the number of ones, the rank and unrank algorithms can be extended to the inverted bitvector representation of a state.

The correctness argument relies on the binomial coefficients labeling a grid graph of nodes $B_{i,j}$ with i denoting the position in the bitvector and j denoting the number of 0s already seen. Let $B_{i,j}$ be connected via a directed edge to $B_{i-1,j}$ and $B_{i-1,j-1}$ corresponding to a 1 and a 1 processed in the bitvector. Starting at $B_{i,j}$ there are $\binom{i}{j}$ possible non-overlapping paths that reach $B_{0,z}$. These pathcount-values can be used to determine the index of a given bitvector in the set of all possible ones. At the current node (i,j) in the grid graph in case of the state at position i containing a 1: all path-counts at $B_{i-1,j-1}$ are added; or a 0: nothing is added.

Hashing with Multinomial Coefficients

The perfect hash functions derived for games like *Peg-Solitaire* are often insufficient in games with pieces of different color like *TicTacToe* and *Nine-Men's-Morris*. For this case, we devise a hash function that operates on state vectors of size n that contain 0s (location not occupied), ones (location occupied by pieces of the first player) and twos (location occupied by pieces of the second player). We will determine the value of a position by hashing all state with a fixed number of z zeros, and o ones and $t = n-z-o$ twos to a value in $\{0,\ldots,\binom{n}{z,o,t}-1\}$, where the multinomial coefficient $\binom{n}{z,o,t}$ is defined as

$$\binom{n}{z,o,t} = \frac{n!}{z!\cdot o!\cdot t!}.$$

The correctness argument relies on representing the multinomial coefficients in a 3D grid graph of nodes $B_{i,j,l}$ with i denoting the index position in the vector and j denoting the number of zeros j, and l denoting the number of ones already seen. The number of twos is then immediate. Let $B_{i,j,l}$ be connected via a directed edge to $B_{i-1,j,l}$, $B_{i-1,j,l-1}$ and $B_{i-1,j-1,l}$ corresponding to a value 2, 1 and 0 processed in the bitvector, respectively. There are $\binom{i}{j,l,n-j-l}$ possible non-overlapping paths starting from each node $B_{i,j,l}$ that reach $B_{0,z,o}$. These pathcount-values can be used to determine the index of a given bitvector in the set of all possible ones. At the current node (i,j,l) in the grid graph in case of the node at position i containing a 1: all path-counts values at $B_{i-1,j-1,l}$ are added; a 2: all path-counts values at $B_{i-1,j,l-1}$ are added; or a 0: nothing is added.

12.6 Parallelization

Parallel processing is the future of computing. On current personal computer systems with multiple cores on the CPU and (graphics) processing units on the graphics card, parallelism is available "for the masses". For the case of solving games, we aim at fast successor computation. Moreover, ranking and unranking that take substantial running time are executed in parallel.

To improve the I/O behavior, the partitioned state space was distributed over multiple hard disks. This increased the reading and writing bandwidth and enabled each thread to use its own hard disk. In larger instances that exceed RAM capacities we additionally maintain write buffers to avoid random access on disk. Once the buffer is full, it is flushed to disk. In one streamed access, all corresponding bits are set.

Multi-Core Computation

Nowadays computers have multiple cores, which reduce the runtime of an algorithm by distributing the workload to concurrently running threads. We use *pthreads* for such multi-threading support.

Let S_p be the set of all possible positions in *Fox-and-Geese* (*Frogs-and-Toads*) with p pieces, which together with the fox position and the player's turn uniquely address states in the game. During play, the number of pieces decreases (or stays) such that we partition backward (forward) BFS layers into disjoint sets $S_p = S_{p,0} \cup \ldots \cup S_{p,n-1}$. As $|S_{p,i}| \leq \binom{n-1}{p}$ is constant for all $i \in \{0,\ldots,n-1\}$, a possible upper bound on the number of reachable states with p pieces is $n \cdot \binom{n-1}{p}$. These states will be classified by the algorithm.

In two-bit retrograde (bfs) analysis all layers $Layer_0, Layer_1, \ldots$ are processed in partition form. The fixpoint iteration to determine the solvability status in one backward (forward) BFS level $Layer_p = S_{p,0} \cup \ldots \cup S_{p,n-1}$ is the most time consuming part. Here, we can apply a multi-core parallelization using pthreads. In total, n threads are forked and joined after completion. They share the same hash function and communicate for termination.

For improving space consumption we urge the exploration to flush the sets $S_{p,i}$ whenever possible and to load only the ones needed for the current computation. In the retrograde analysis of *Fox-and-Geese* the access to positions with a smaller number of pieces S_{p-1} is only needed during the initialization phase. As such initialization is a simple scan through a level we only need one set $S_{p,i}$ at a time. To save space for the fixpoint iteration, we release the memory needed to store the previous layer. As a result, the maximum number of bits needed is $\max\{|S_p|, |S_p|/n + |S_{p-1}|\}$.

GPU Computation

In the last few years there has been a remarkable increase in the performance and capabilities of the graphics processing unit. Modern GPUs are not only powerful, but also parallel programmable processors featuring high arithmetic capabilities and memory bandwidths. Deployed on current graphic cards, GPUs have outpaced CPUs in many numerical algorithms. The GPU's rapid increase in both programmability and capability has inspired researchers to map computationally demanding, complex problems to the GPU.

GPUs have multiple cores, but the programming and computational model are different from the ones on the CPU. Programming a GPU requires a special compiler, which translates the code to native GPU instructions. The GPU architecture mimics a single instruction multiple data (SIMD) computer with the same instructions running on all processors. It supports different layers for memory access, forbids simultaneous writes but allows concurrent reads to one memory cell.

Memory, is structured hierarchically, starting with the GPU's global memory (video RAM, or VRAM). Access to this memory is slow, but can be accelerated through *coalescing*, where adjacent accesses with less than 64 bits are combined to one 64-bit access. Each SM includes 16 KB of memory (SRAM), which is shared between all SPs and can be accessed at the same speed as registers. Additional registers are also located in each SM but not shared between SPs. Data must be copied from the systems main memory to the VRAM to be accessible by the threads.

The GPU programming language links to ordinary C-sources. The function executed in parallel on the GPU is called *kernel*. The kernel is driven by threads, grouped together in *blocks*. The TSC distributes the blocks to its SMs in a way that none of them runs more than *maxThreads* threads and a block is not distributed among different SMs. This way, considering that the maximal *blockSize*, at most $\lceil maxThreads/blockSize \rceil$ blocks can be executed by one SM on its SPs. Each SM schedules the threads (one for each SP) to be executed in parallel, providing the code to the SPs. Since all the SPs get the same chunk of code, SPs in an else-branch wait for the SPs in the if-branch, being idle. After the eight threads have completed a chunk the next one is executed. Note that threads waiting for data can be parked by the SM, while the SPs work on threads, which have already received the data.

To profit from coalescing, threads should access adjacent memory contemporary. Additionally, the SIMD-like architecture forces us to avoid if-branches and to design a kernel which will be executed unchanged for all threads. These facts lead to the implementation of keeping the entire or partitioned state space bitvector in RAM and copying an array of indices (ranks) to the GPU. This approach benefits from the SIMD technology but imposes additional work on the CPU. One additional scan through the bitvector is needed to convert its bits into integer ranks, but on the GPU the work to unrank, generate the successors and rank them is identical for all threads. To avoid unnecessary memory access, the rank given to expand should be overwritten with the rank of the first child. As the number of successors is known beforehand, with each rank we reserve space for its successors. For smaller BFS layers this means that fewer states are expanded.

For solving games on the GPU, storing the bitvector on the GPU yields bad exploration results. Hence, we forward the bitvector indices from the CPU's host RAM to the GPU's VRAM, where they were uploaded to the SRAM, unranked and expanded, while the successors were ranked. At the end of one iteration, all successors are moved back to CPU's host RAM, where they are perfectly hashed and marked if new.

12.7 Experiments Explicit-State Perfect Hashing

We start with the presentation of the experiments in permutation games (mostly showing the effect of multi-core GPU computation) followed by selection games (also showing the effect of multi-core CPU computation). For measuring the speed-up on a matching implementation, we compare the GPU performance with a CPU emulation on a single core. This way, the same code and work was executed on the CPU and the GPU. For a fair comparison, the emulation was run with GPU code adjusted to one thread. This minimizes the work for thread communication on the CPU. Moreover, we profiled that the emulation consumed most CPU time for state expansion and ranking.

Sliding-Tile Puzzle

The results of the first set of experiments shown in Table 12.1 illustrate the effect of bitvector state space compression with breadth-first search in rectangular *Sliding-Tile* problems of different sizes.

We run both the one- and two-bit breadth-first search algorithms on the CPU and GPU. The 3×3 version was simply too small to show significant advances, while even in partitioned form a complete exploration on a bit vector representation of the 15-Puzzle requires more RAM than available.

We first validated that all states were generated and equally distributed among the possible blank positions. Moreover, as expected, the numbers of BFS layers for symmetric puzzle instances match (53 for 3×4 and 4×3 as well as 63 for 2×6 and 6×2).

For the 2-Bit BFS implementation, we observe a moderate speed-up by a factor between 2 and 3, which is due to the fact that the BFS-layers of the instances that could be solved in RAM are too small. For such small BFS layers, further data processing issues like copying the indices to the VRAM is rather expensive compared to the gain achieved by parallel computation on the GPU. Unfortunately, the next larger instance (7×2) was too large for the amount of RAM available in the machine (it needs $3 \times 750 = 2,250$ MB for *Open* and 2 GB for reading and writing indices to the VRAM).

In the 1-Bit BFS implementation the speed-up increases to a factor between 7 and 10 in the small instances. Many states are re-expanded in this approach, inducing more work for the GPU and exploiting its potential for parallel computation. Partitions being too large for the VRAM are split and processed in chunks of about 250 million indices (for the 7×2 instance). A quick calculation shows that the savings of GPU computation are large. We noticed that the GPU has the capability to generate 83 million states per second (including unranking, generating the successors and computing their rank) compared to about five million states per second of the CPU. As a result, for the CPU experiment that ran out of time (o.o.t), which we stopped after one day of execution, we predict a speed-up factor of at least 16, and a running time of over 60 hours.

Problem	2-Bit Time GPU	2-Bit Time CPU	1-Bit Time GPU	1-Bit Time CPU
(2×6)	1m10s	2m56s	2m43s	15m17s
(3×4)	55s	2m22s	1m38s	13m53s
(4×3)	1m4s	2m22s	1m44s	12m53s
(6×2)	1m26s	2m40s	1m29s	18m30s
(7×2)	o.o.m.	o.o.m.	226m30s	o.o.t.

Table 12.1: Comparing GPU with CPU performances in 1-Bit and 2-Bit BFS in sliding-tile puzzles.

n	States	GPU Time	CPU Time
6	120	0s	0s
7	360	0s	0s
8	5,040	0s	0s
9	20,160	0s	0s
10	362,880	0s	6s
11	1,814,400	1s	35s
12	39,916,800	27s	15m20s

Table 12.2: Comparing GPU with CPU performances for Two-Bit-BFS in the Top-Spin.

n	States	GPU Time	CPU Time
9	362,880	0s	4s
10	3,628,800	2s	48s
11	39,916,800	21s	10m41s
12	479,001,600	6m50s	153m7s

Table 12.3: Comparing GPU with CPU performances in Two-Bit-BFS in Pancake problems.

Top-Spin Problems

The results for the (n,k)-Top-Spin problems for a fixed value of $k=4$ are shown in Table 12.2 (o.o.m. denotes out of memory, while o.o.t. denotes out of time). We see that the experiments validate the theoretical statement that the state spaces are of size $(n-1)!/2$ for n odd and $(n-1)!$ for n even. For large values of n, we obtain a significant speed-up of more than factor 30.

Pancake Problems

The GPU and CPU running time results for the n-Pancake problems are shown in Table 12.3. Similarly to the Top-Spin puzzle for a large value of n, we obtain a speed-up factor of more than 30 with respect to running the same algorithm on the CPU.

Peg-Solitaire

The first set of results, shown in Table 12.4, considers *Peg-Solitaire*. For each BFS-layer, the state space is small enough to fit in RAM. The exploration result show that there are five positions with one peg remaining (of course, there is none with zero pegs), one of which has the peg in the goal position.

In *Peg-Solitaire* we find a symmetry, which applies to the entire state space. If we invert the board (exchanging pegs with holes or swapping the colors), the goal and the initial state are the same. Moreover, the entire forward and backward graph structures match.

Hence, a call of backward breadth-first search to determine the number of states with a fixed goal distance is not needed. The number of states for a given goal distance matches the number of states with the same distance to the initial state. The total number of reachable states is 187,636,298.

We parallelized the game expanding and ranking states on the GPU. The total time for a BFS we measured was about 12 minutes on the CPU and 1 minute on the GPU. As the puzzle is moderately small, we consider the speed-up factor with respect to CPU computation to be significant.

Holes	Bits	Space	Expanded
0	1	1 B	–
1	33	5 B	1
2	528	66 B	4
3	5,456	682 B	12
4	40,920	4,99 KB	60
5	237,336	28,97 KB	296
6	1,107,568	135 KB	1,338
7	4,272,048	521 KB	5,648
8	13,884,156	1.65 MB	21,842
9	38,567,100	4.59 MB	77,559
10	92,561,040	11.03 MB	249,690
11	193,536,720	23.07 MB	717,788
12	354,817,320	42.29 MB	1,834,379
13	573,166,440	68.32 MB	4,138,302
14	818,809,200	97.60 MB	8,171,208
15	1,037,158,320	123 MB	14,020,166
16	1,166,803,110	139 MB	20,773236
17	1,166,803,110	139 MB	26,482,824
18	1,037,158,320	123 MB	28,994,876
19	818,809,200	97.60 MB	27,286,330
20	573,166,440	68.32 MB	22,106,348
21	354,817,320	42.29 MB	15,425,572
22	193,536,720	23.07 MB	9,274,496
23	92,561,040	11.03 MB	4,792,664
24	38,567,100	4.59 MB	2,120,101
25	13,884,156	1.65 MB	800,152
26	4,272,048	521 KB	255,544
27	1,107,568	135 KB	68,236
28	237,336	28.97 KB	14,727
29	40,920	4.99 KB	2529
30	5,456	682 B	334
31	528	66 B	33
32	33	5 B	5
33	1	1 B	-

Table 12.4: Applying One-Bit-BFS to *Peg-Solitaire*.

The exploration results match with the ones in the general game player. For this case we had to alter the reward structure to the one that is imposed by the general game description language that was used there. We found that the number of expanded states matches, but – as expected – the total time to classify the states using the specialized player on the GPU is much smaller than in the general player running on one core of the CPU.

Frogs-and-Toads

Similar to *Peg-Solitaire* if we invert the board (swapping the colors of the pieces), the goal and the initial state are the same, so that forward breadth-first search suffices to solve the game. The result for *Fore and Aft* that reversing black and white takes 46 moves is easily validated with BFS. There are two positions which require 47 moves, namely, after reversing black and white, putting one of the far corner pieces in the center. Table 12.5 also shows that there are 218,790 positions in total.

As *Frogs-and-Toads* generalizes *Fore and Aft*, we next considered the variant with 15 black and 15 white pieces on a board with 31 squares. The BFS outcome is shown in Table 12.6. We monitored that reversing black and white pieces takes 115 steps (in a shortest solution) and see that the worst-case input is slightly harder and takes 117 steps. A GPU parallelization leading to the same exploration results required about half an hour run-time.

Depth	Expanded	Depth	Expanded	Depth	Expanded	Depth	Expanded
1	1	13	1,700	25	15,433	37	1,990
2	8	14	2,386	26	14,981	38	1,401
3	13	15	3,223	27	14,015	39	914
4	14	16	4,242	28	12,848	40	557
5	32	17	5,677	29	11,666	41	348
6	58	18	7,330	30	10,439	42	202
7	121	19	8,722	31	9,334	43	137
8	178	20	10,084	32	7,858	44	66
9	284	21	11,501	33	6,075	45	32
10	494	22	12,879	34	4,651	46	4
11	794	23	13,997	35	3,459	47	11
12	1,143	24	14,804	36	2,682	48	2

Table 12.5: BFS Results for *Fore and Aft*.

Depth	Expanded	Depth	Expanded	Depth	Expanded	Depth	Expanded
1	1	31	4,199,886	61	171,101,874	91	4,109,157
2	8	32	5,447,660	62	170,182,837	92	3,156,288
3	17	33	6,975,087	63	168,060,816	93	2,387,873
4	26	34	8,865,648	64	164,733,845	94	1,780,521
5	46	35	11,138,986	65	160,093,746	95	1,307,312
6	78	36	13,881,449	66	154,297,247	96	948,300
7	169	37	17,060,948	67	147,342,825	97	680,299
8	318	38	20,800,347	68	139,568,855	98	484,207
9	552	39	25,048,652	69	131,146,077	99	340,311
10	974	40	29,915,082	70	122,370,443	100	235,996
11	1,720	41	35,382,942	71	113,415,294	101	160,153
12	2,905	42	41,507,233	72	104,380,748	102	107,024
13	4,826	43	48,277,767	73	95,379,850	103	69,216
14	7,878	44	55,681,853	74	86,375,535	104	44,547
15	12,647	45	63,649,969	75	77,534,248	105	27,873
16	19,980	46	72,098,327	76	68,891,439	106	17,394
17	31,511	47	80,937,547	77	60,672,897	107	10,256
18	49,242	48	89,999,613	78	52,953,463	108	6,219
19	74,760	49	99,231,456	79	45,889,798	109	3,524
20	112,218	50	108,495,904	80	39,482,737	110	2,033
21	166,651	51	117,679,229	81	33,751,896	111	1,040
22	241,157	52	126,722,190	82	28,607,395	112	532
23	348,886	53	135,363,894	83	24,035,844	113	251
24	497,698	54	143,534,546	84	19,957,392	114	154
25	700,060	55	150,897,878	85	16,394,453	115	42
26	974,219	56	157,334,088	86	13,306,659	116	19
27	1,337,480	57	162,600,933	87	10,695,284	117	10
28	1,812,712	58	166,634,148	88	8,521,304	118	2
29	2,426,769	59	169,360,939	89	6,738,557		
30	3,214,074	60	170,829,205	90	5,286,222		

Table 12.6: BFS Results for *Frogs-and-Touds*.

Fox-and-Geese

The next set of results shown in Table 12.7 considers the *Fox-and-Geese* game, where we applied retrograde analysis. For a fixed fox position the remaining geese can be binomially hashed.

The first three levels do not contain any state won for the geese, which matches the fact that four geese are necessary to block the fox (at the middle border cell in each arm of the cross). We observe that after a while, the number of iterations shrinks for a rising number of geese. This matches the experience that with more geese it is easier to block the fox.

Recall that all positions that couldn't been proven won or lost by the geese are devised to be a win for the fox. The critical point, where the fox loses more than 50% of the game is reached at level 16. This matches the observation in practical play, that the 13 geese are too few to show an edge for the geese.

The total run-time of about a month for the experiment is considerable. Without multi-core parallelization, however, more than seven months would have been needed to complete the experiments. Even though we paralellized only the iteration stage of the algorithm, the speed-up on the 4-core hyper-threaded machine is larger than 7, showing an almost linear speed-up.

The total of space needed for operating an optimal player is about 34 GB, so that in case geese are captured we would have to reload data from disk. This strategy yields a maximal space requirement of 4.61 GB RAM, which might further be reduced by reloading data in case a fox moves.

Geese	States	Space	Iterations	Won	Time Real	Time User
1	2,112	264 B	1	0	0.05s	0.08s
2	32,736	3.99 KB	6	0	0.55s	1.16s
3	327,360	39 KB	8	0	0.75s	2.99s
4	2,373,360	289 KB	11	40	6.73s	40.40s
5	13,290,816	1.58 MB	15	1,280	52.20s	6m24s
6	59,808,675	7.12 MB	17	21,380	4m37s	34m40s
7	222,146,996	26 MB	31	918,195	27m43s	208m19s
8	694,207,800	82 MB	32	6,381,436	99m45s	757m0s
9	1,851,200,800	220 MB	31	32,298,253	273m56s	2,083m20s
10	4,257,807,840	507 MB	46	130,237,402	1,006m52s	7,766m19s
11	8,515,615,680	1015 MB	137	633,387,266	5,933m13s	46,759m33s
12	14,902,327,440	1.73 GB	102	6,828,165,879	4,996m36s	36,375m09s
13	22,926,657,600	2.66 GB	89	10,069,015,679	5,400m13s	41,803m44s
14	31,114,749,600	3.62 GB	78	14,843,934,148	5,899m14s	45,426m42s
15	37,337,699,520	4.24 GB	73	18,301,131,418	5,749m6s	44,038m48s
16	39,671,305,740	4.61 GB	64	20,022,660,514	4,903m31s	37,394m1s
17	37,337,699,520	4.24 GB	57	19,475,378,171	3,833m26s	29,101m2s
18	31,114,749,600	3.62 GB	50	16,808,655,989	2,661m51s	20,098m3s
19	22,926,657,600	2.66 GB	45	12,885,372,114	1,621m41s	12,134m4s
20	14,902,327,440	1.73 GB	41	8,693,422,489	858m28s	6,342m50s
21	8,515,615,680	1015 MB		5,169,727,685	395m30s	2,889m45s
22	4,257,807,840	507 MB	31	2,695,418,693	158m41s	1,140m33s
23	1,851,200,800	220 MB	26	1,222,085,051	54m57	385m32s
24	694,207,800	82 MB	23	477,731,423	16m29s	112m.35s
25	222,146,996	26 MB	20	159,025,879	4m18s	28m42s
26	59,808,675	7.12 MB	17	44,865,396	55s	5m49s
27	13,290,816	1.58 MB	15	10,426,148	9.81s	56.15s
28	2,373,360	289 KB	12	1,948,134	1.59s	6.98s
29	327,360	39 KB	9	281,800	0.30s	0.55s
30	32,736	3.99 KB	6	28,347	0.02s	0.08s
31	2,112	264 B	5	2001	0.00s	0.06s

Table 12.7: Retrograde analysis results for *Fox-and-Geese*.

Symmetries, Frontier Search, and Generality

In many board games we find symmetries like reflection along the main axes or along the diagonals. If we look at the four possible rotations on the board for *Peg-Solitaire* and *Fox-and-Geese* plus reflection, we count eight symmetries in total. For *Fox-and-Geese* we can classify all states that share a symmetrical fox position by simply copying the result obtained for the existing one. Besides the savings of time for not expanding states, this can also save the number of positions that have to be kept in RAM during fixpoint computation. If the forward and backward search graphs match (as in *Peg Solitaire* and *Frogs-and-Toads*) we may also truncate the breadth-first search procedure to the half of the search depth. In two-bit BFS, we simply have to look at the rank

of the inverted unranked state. Moreover, with the forward BFS layers we also have the minimal distances of each state to the goal state, and, hence, the classification result.

Frontier search is motivated by the attempt of omitting the *Closed* list of states already expanded. It mainly applies to problem graphs that are directed or acyclic but has been extended to more general graph classes. It is especially effective if the ratio of *Closed* to *Open* list sizes is large. Frontier search requires the *locality* of the search space being bounded, where the locality (for breadth-first search) is defined as $\max\{layer(s) - layer(s') + 1 \mid s, s' \in S; s' \in succs(s)\}$, where $layer(s)$ denotes the depth d of s in the breadth-first search tree. For frontier search, the *space efficiency* of the hash function $h : S \to \{0, \ldots, m-1\}$ boils down to $m/(\max_d |Layer_d| + \ldots + |Layer_{d+l}|)$, where $Layer_d$ is set of nodes in depth d of the breadth-first search tree and l is the locality of the breadth-first search tree as defined above.

For the example of the Fifteen puzzle, i.e., the 4×4 version of *Sliding-Tile*, the predicted amount of 1.2 TB hard disk space for 1-bit breadth-first search is only slightly smaller than the 1.4 TB of frontier breadth-first search. As frontier search does not shrink the set of states reachable, one may conclude, that frontier search hardly cooperates well with a bitvector representation of the entire state space. However, if layers are hashed individually, as done in all selection games we have considered, a combination of bitstate and frontier search is possible.

Compressed Pattern Databases

The number of bits per state can be reduced to $\log 3 \approx 1.6$. For this case, five values $\{0, 1, 2\}$ are packed into a byte, given that $3^5 = 243 < 255$.

The idea of pattern database compression is to store the mod-3 value (of the backward BFS depth) from abstract space, so that its absolute value can be computed incrementally in constant time. For the initial state, an incremental computation for its heuristic evaluation is not available, so that a backward construction of its generating path can be used. For an undirected graph a shortest path predecessor with mod-3 of BFS depth k appears in level $k-1$ *mod* 3.

As the abstract space is generated anyway for generating the database, one could alternatively invoke a shortest path search from the initial state, without exceeding the time complexity of database construction.

By having computed the heuristic value for the projected initial state as the goal distance in the inverted abstract state space graph, all other pattern database lookup values can then be determined incrementally in constant time, i.e., $h(v) = h(u) + \Delta(v)$, with $v \in succs(u)$ and $\Delta(v)$ found using the mod-3 value of v. If the considered search spaces are undirected, the information to evaluate the successors with $\Delta(v) \in \{-1, 0, 1\}$ is possible.

For directed (and unweighted) search spaces more bits are needed to allow incremental heuristic computation in constant time. It is not difficult to see that the locality in the inverted abstract state space determines the maximum difference in h-values $h(v) - h(u)$, $v \in succs(u)$ in the original space.

In a directed (but unweighted) search space, the (dual) logarithm of the (breadth-first) locality of the inverse of the abstract state space graph plus 1 is an upper bound on the number of bits needed for incremental heuristic computation of bitvector compressed pattern databases, i.e., for locality $l_A^{-1} = \max\{layer^{-1}(u) - layer^{-1}(v) + 1 \mid u, v \in A; v \in succs^{-1}(u)\}$ in abstract state space graph A of S we require at most $\log\lceil l_A^{-1} \rceil + 1$ bits to reconstruct the value $h(v)$ of a successor $v \in S$ of any chosen $u \in S$ given $h(u)$.

First we observe that the goal distances in abstract space A determine the h-value in the original state space, so that the locality $\max\{layer^{-1}(u) - layer^{-1}(v) + 1 \mid u, v \in A; v \in succs^{-1}(u)\}$ is bounded by $h(u) - h(v) + 1$ for all u, v in original space with $u \in succs(v)$, which is equal to the maximum of $h(v) - h(u) + 1$ for $u, v \in S$ with $v \in succs(u)$. Therefore, the number of bits needed for incremental heuristic computation equals $\lceil \max\{h(v) - h(u) \mid u, v \in A; v \in succs^{-1}(u)\} \rceil + 2$ as all values in the interval $[h(u) - 1, \ldots, h(v)]$ have to be accommodated. Thus, for the incremental value $\Delta(v)$ added to $h(u)$ we have $\Delta(v) \in \{-1, \ldots, h(v) - h(u)\}$, so that $\lceil \log(\max\{h(v) - h(u) + 2 \mid u, v \in S; v \in succs(u)\}) \rceil = \log\lceil l_A^{-1} \rceil + 1$ bits suffice to reconstruct the value $h(v)$ of a successor $v \in S$ for every $u \in S$ given $h(u)$.

For undirected search spaces we have $\log l_A^{-1} = \log 2 = 1$, so that $1+1=2$ bits suffice to be stored for each abstract pattern state according to the theorem. Using the tighter packing of the $2+1=3$ values into bytes provided above, $8/5 = 1.6$ bits are sufficient.

If not all states in the search space that has been encoded in the perfect hash function are reachable, reducing the constant-bit compression to a lesser number of bits might not always be available, as unreached states cannot easily be removed. For this case, the numerical value remaining to be set for an unreachable state in the inverse of the abstract state space, will stand for h-value infinity, at which the search in the original search space can stop.

More formally, the best-first locality has been defined as $\max\{\mathit{cost\text{-}layer}(s) - \mathit{cost\text{-}layer}(s') + \mathit{cost}(s,s') \mid s,s' \in S; s' \in \mathit{succs}(s)\}$, where $\mathit{cost\text{-}layer}(s)$ denotes the smallest accumulated *cost*-value from the initial state to s. The theoretical considerations on the number of bits needed to perform incremental heuristic evaluation extend to this setting.

Other Games

In *Rubik's Cube* each face can be rotated by 90, 180, or 270 degrees and the goal is to rearrange a scrambled cube such that all faces are uniformly colored. Solvability constraints for the set of all dissembled cubes are: a single corner cube must not be twisted; a single edge cube must not be twisted and no two cube must be exchanged. For the last item the parity of the permutation is crucial and leads to $8! \cdot 3^7 \cdot 12! \cdot 2^{11}/2 \approx 4.3 \cdot 10^{19}$ states. Assuming one bit per state, an impractical number of $4.68 \cdot 10^{18}$ bytes for performing full reachability is needed.

The binomial and multinomial hashing approach is applicable to many other games.

- In *Awari* the two players redistribute seeds among 12 holes according to the rules of the game, with an initial state having uniformly four seeds in each of the holes. When all seeds are available, all possible layouts can be generated in an urn experiment with 59 balls, where 48 balls represent filling the current hole with a seed and 11 balls indicate changing from the current to the next hole. Thus, binomial hashing applies.
- In *Dots and Boxes* players take turns joining two horizontally or vertically adjacent dots by a line. A player that completes the fourth side of a square (a box) colors that box and must play again. When all boxes have been colored, the game ends and the player who has colored more boxes wins. Here, the binomial hash suffices. For each edge we denote whether it is marked. Together with the marking, we denote the number of boxes of at least one player. In contrast to other games, all successor share in the next layer, so that one scan suffices to solve the current one.
- *Nine-Men's-Morris* is one of the oldest games still played today. The game naturally divides into three stages. Each player has nine pieces, called men, that are first placed alternately on a board with 24 locations. In the second stage, the men move to form mills (a row of three pieces along one of the board's lines), in which case one man of the opponent (except the ones that form a mill) is removed from the board. In one common variation of the third stage, once a player is reduced to three men, his pieces may "fly" to any empty location. If a move has just closed a mill, but all the opponent's men are also in mills, the player may declare any stone to be removed. The game ends if a player has less than three men (the player loses), if a player cannot make a legal move (the player loses), if a midgame or endgame position is repeated (the game is a draw). Besides the usual symmetries along the axes, there is one in swapping the inner with the outer circle. For this game, multinomial hashing is applicable.

12.8 Binary Decision Diagrams for Strongly Solving Games

In general state spaces one can derive minimal perfect hash functions with a few bits per state (I/O-efficiently) *after* generating the state space (on disk). This approach applies hypergraph theory, which exceeds the scope of the book. It requires c bit RAM per state (typically, $c \approx 2$). Of course, perfect hash functions do not have to be

minimal to be space efficient. Non-minimal hash functions can outperform minimal ones since the gain in the constant c for the hash function can be more important than the loss in coverage.

Binary decision diagrams (BDDs) are a memory-efficient data structure used to represent Boolean functions. A BDD is a directed acyclic graph with one root and two terminal nodes, the 0- and the 1-sink. Each internal node corresponds to a binary variable and has two successors, one (along the *Then*-edge) representing that the current variable is true (1) and the other (along the *Else*-edge) representing that it is false (0). For any assignment of the variables derived from a path from the root to the 1-sink the represented function will be evaluated to 1.

For a fixed variable ordering, two reduction rules can be applied: eliminating nodes with the same Then and Else successors and merging two nodes representing the same variable that share the same Then successor as well as the same Else successor. These BDDs are called reduced ordered binary decision diagrams (ROBDDs). Whenever we mention BDDs, we refer to ROBDDs. We also assume that the variable ordering is the same for all the BDDs and has been optimized prior to the search.

We are interested in the *image* of a state set S with respect to a transition relation *Trans*. The result is a characteristic function of all states reachable from the states in S in one step. For the application of the image operator we need two sets of variables, one, x, representing the current state variables, another, x', representing the successor state variables. The image *Succ* of the state set S is then computed as $Succ(x') = \exists x\ (Trans(x,x') \wedge S(x))$. The *preimage Pre* of the state set S is computed as $Pre(x) = \exists x'\ (Trans(x,x') \wedge S(x'))$ and results in the set of predecessor states.

Using the image operator, implementing a layered symbolic breadth-first search (BFS) is straight-forward. All we need to do is to apply the image operator to the initial state resulting in the first layer, then apply the image operator to the first layer resulting in the second and so on. The search ends when no successor states can be found. General games (and in this case, *Connect Four*) are guaranteed to terminate after a finite number of steps, so that the forward search will eventually terminate as well.

The problem of the construction of perfect hash functions for algorithms like *two-bit breadth-first search* is that most of them are problem-dependent. Hence, for the construction of the perfect hash function, the underlying state set to be hashed is generated in advance in form of a BDD. This is true, when computing strong solutions to problems, where we are interested in the game-theoretical value of all reachable states. Applications are, e.g., endgame databases or planning tasks where the problem to be solved is harder than computing the reachability set.

The $index(n)$ of a BDD node n is its unique position in the shared representation and $level(n)$ its position in the variable ordering. Moreover, we assume the 1-sink to have index 1 and the 0-sink to have index 0. Let $C_f = |\{a \in \{0,1\}^n \mid f(a) = 1\}|$ denote the number of satisfying assignments (*satcount*, here also *sc* for short) of f. With *bin* (and *invbin*) we denote the *conversion* of the binary value of a bitvector (and its inverse). The *rank* of a satisfying assignment $a \in \{0,1\}^n$ is the position in the lexicographical ordering of all satisfying assignments, while the *unranking* of a number r in $\{0,\ldots,C_f - 1\}$ is its inverse.

Figure 12.5 shows the ranking and unranking functions in pseudo-code. The procedures determine the rank given a satisfying assignment and vice versa. They access the satcount values on the Else-successor of each node (adding for the ranking and subtracting in the unranking). Missing nodes (due to BDD reduction) have to be accounted for by their binary representation, i.e., gaps of l missing nodes are accounted for as 2^l. While the ranking procedure is recursive the unranking procedure is not.

The satcount values of all BDD nodes are precomputed and stored along with the nodes. As BDDs are reduced, not all variables on a path are present but need to be accounted for in the satcount procedure. The time (and space) complexity of it is $O(|G_f|)$, where $|G_f|$ is the number of nodes of the BDD G_f representing f. With the precomputed values, rank and unrank both require linear time $O(n)$, where n is the number of variables in the function represented in the BDD.

To illustrate the ranking and unranking procedures, take the example BDD given in Figure 12.6. Assume we want to calculate the rank of state $s = 110011$. The rank of s is then

```
rank(s)
  i = level(root);
  d = bin(s[0..i-1]);
  return d*sc(root) + rankAux(root,s) - 1;
```

```
rankAux(n, s)
  if (n <= 1) return n;
  i = level(n);
  j = level(Else(n));
  k = level(Then(n));
  if (s[i] == 0)
    return bin(s[i+1..j-1]) * sc(Else(n))
      + rankAux(Else(n),s);
  else
    return 2^(j-i-1) * sc(Else(n))
      + bin(s[i+1..k-1]) * sc(Then(n))
      + rankAux(Then(n),s);
```

```
unrank(r)
  i = level(root);
  d = r / sc(root);
  s[0..i-1] = invbin(d);
  n = root;
  while (n > 1)
    r = r mod sc(n);
    j = level(Else(n));
    k = level(Then(n));
    if (r < (2^(j-i-1) * sc(Else(n))))
      s[i] = 0;
      d = r / sc(Else(n));
      s[i+1..j-1] = invbin(d);
      n = Else(n);
      i = j;
    else
      s[i] = 1;
      r = r - (2^(j-i-1) * sc(Else(n)));
      d = r / sc(Then(n));
      s[i+1..k-1] = invbin(d);
      n = Then(n);
      i = k;
  return s;
```

Figure 12.5: Ranking and unranking.

$$\begin{aligned}
rank(s) = 0 + rA(v_{13},s) - 1 &= (2^{1-0-1} \cdot sc(v_{11}) + 0 + rA(v_{16},s)) - 1 \\
&= sc(v_{11}) + (2^{3-1-1} \cdot sc(v_8) + bin(0) \cdot sc(v_9) + rA(v_9,s)) - 1 \\
&= sc(v_{11}) + 2sc(v_8) + (0 + rA(v_5,s)) - 1 \\
&= sc(v_{11}) + 2sc(v_8) + (2^{6-4-1} \cdot sc(v_0) + bin(1) \cdot sc(v_1) + rA(v_1,s)) - 1 \\
&= sc(v_{11}) + 2sc(v_8) + 2sc(v_0) + sc(v_1) + 1 - 1 \\
&= 14 + 2 \cdot 5 + 2 \cdot 0 + 1 + 1 - 1 = 25,
\end{aligned}$$

with $rA(s,v_i)$ being the recursive call of the rankAux function for state s in node v_i and $sc(v_i)$ the satcount stored in node v_i.

For unranking the state with index 19 ($r = 19$) from the BDD depicted in Figure 12.6 we get:

- $i = 0, n = v_{13}$: $r = 19 \mod sc(v_{13}) = 19 \mod 30 = 19 \not< 2^{1-0-1} sc(v_{11}) = 14$, thus $s[0] = 1$; $r = r - 2^{1-0-1} sc(v_{11}) = 19 - 14 = 5$
- $i = 1, n = v_{12}$: $r = 5 \mod sc(v_{12}) = 5 \mod 16 = 5 < 2^{3-1-1} sc(v_8) = 2 \cdot 5 = 10$, thus $s[1] = 0$; $s[2] = invbin(r/sc(v_8)) = invbin(5/5) = 1$
- $i = 3, n = v_8$: $r = 5 \mod sc(v_8) = 5 \mod 5 = 0 < 2^{4-3-1} sc(v_4) = 1$, thus $s[3] = 0$
- $i = 4, n = v_4$: $r = 0 \mod sc(v_4) = 0 \mod 1 = 0 \not< 2^{6-4-1} sc(v_0) = 0$, thus $s[4] = 1$; $r = r - 2^{6-4-1} sc(v_0) = 0 - 0 = 0$
- $i = 5, n = v_2$: $r = 0 \mod sc(v_2) = 0 \mod 1 = 0 \not< 2^{7-6-1} sc(v_{12}) = 0$, thus $s[5] = 1$; $r = r - 2^{7-6-1} sc(v_{12}) = 0 - 0$
- $i = 6; n = v_1$: return s (= 101011)

Retrograde Analysis on a Bitvector

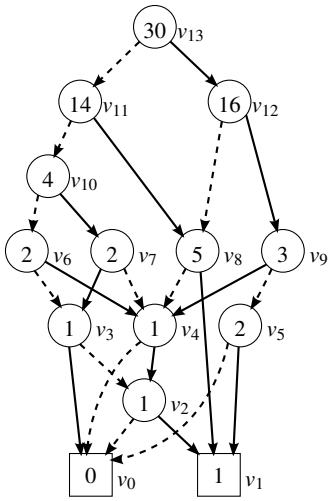


Figure 12.6: BDD for the ranking and unranking examples. Dashed arrows denote Else-edges; solid ones Then-edges. The numbers in the nodes correspond to the satcount. Each v_i denotes the index (i) of the corresponding node.

```
retrograde(won,maxlayers)
for layer in maxlayers,...,0
  m = sc(bdd(layer))
  for i in 0,...,m - 1
    B[layer][i] = 0
  for i in 0,...,m - 1
    state = unrank(i)
    if (won(state))
      if (layer mod 2 == 1)
        B[layer][i] = 1
      else
        B[layer][i] = 2
    else if (layer == maxlayer)
      B[layer][i] = 3
    else
      succs = expand(state)
      process(succs)
```

```
process (succs)
  if (layer mod 2 == 1)
    for all s in succs
      if B[layer+1][rank(s)] == 2
        B[layer][rank(i)] = 2
        break
      else if (B[layer+1][rank(s)]
          == 3)
        B[layer][rank(i)] = 3
    if (B[layer][rank(i)] == 0)
      B[layer][rank(i)] = 1
  else
    for all s in succs
      if B[layer+1][rank(s)] == 1
        B[layer][rank(i)] = 1
        break
      else if (B[layer+1][rank(s)]
          == 3)
        B[layer][rank(i)] = 3
    if (B[layer][rank(i)] == 0)
      B[layer][rank(i)] = 2
```

Figure 12.7: Two-player zero-sum game retrograde analysis (rank and unrank are sensitive to the layer they are called).

In the implementation (see Algorithm 12.7) we also use two bits, but with a different meaning. We apply the algorithm to solve two-player zero-sum games where the outcomes are only won/lost/drawn from the starting player's point of view. This is reflected in the interpretation of the two bits: value 0 means that the state has not yet been evaluated; value 1 means it is won by the starting player (the player with index 0); value 2 means it is won by the player with index 1; value 3 means it is drawn. Retrograde analysis solves the entire set of positions in backward direction, starting from won and lost terminal ones. Bitstate retrograde analysis applies backward BFS starting from the states that are already decided.

For the sake of simplicity, the rank and unrank functions are sensitive with respect to the layer of the search in which the operations take place. In the implementation, we use separate BDDs for the different layers.

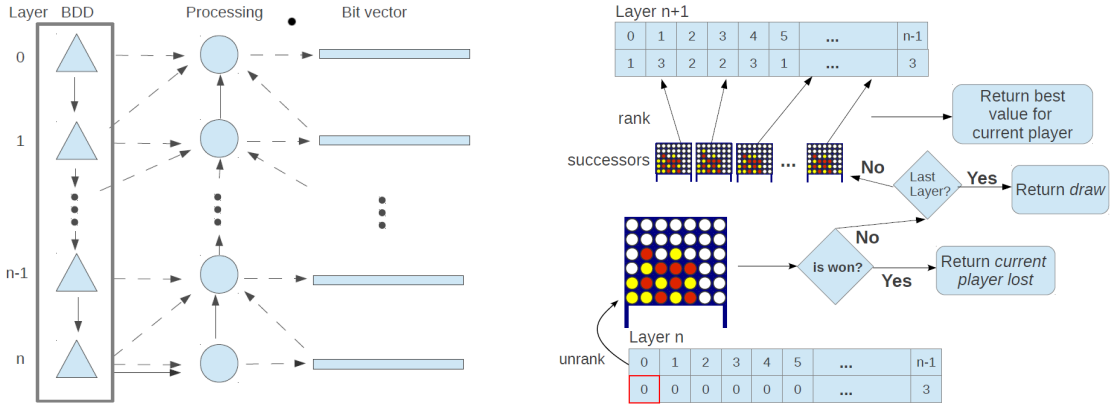


Figure 12.8: Hybrid algorithm: visualization of data flow in the strong solution process (left). Processing a layer in the retrograde analysis (right).

The algorithm assumes a maximal number of moves, that terminal drawn states appear only in the last layer (as is the case in *Connect Four*; extension to different settings is possible), that the game is turn-taking, and that the player can be found in the encoding of the game. It takes as input a decision procedure for determining whether a situation is *won* by one of the players as well as the index of the last reached layer (*maxlayer*). Starting at the final layer, it iterates toward the initial state residing in layer 0.

For each layer, it first determines the number of states. Next it sets all values of the vector B for the states in the current state to 0 – not yet solved. Then it iterates over all states in that layer.

It takes one state (by unranking it from the layer), checks whether it is won by one of the players. If so, it can be solved correspondingly (setting its value to either 1 or 2). Otherwise, if it resides in the final layer, it must be a drawn state (value 3). In case neither holds, we calculate the state's successors. For each successor we check whether it is won by the currently active player, which is determined by checking the current layer's index. In this case the state is assigned the same value and we continue with the next state. Otherwise, if the successor is drawn, the value of this state is set to draw as well. In the end, if the state is still unsolved that means that all successors are won by the opponent, so that the corresponding value is assigned to this state as well.

Hybrid Classification Algorithm

The hybrid classification algorithm combines the two precursing approaches. It generates the state space with symbolic forward search on disk and subsequently applies explicit-state retrograde analysis based on the results in form of the BDD encoded layers read from disk. Figure 12.8 illustrates the strong solution process. On the right-hand side, we see the bitvector used in retrograde analysis and on the left hand side we see the BDD generated in forward search and used in backward search.

The process of solving one layer is depicted in Figure 12.8 (right). While the bitvector in the layer n (shown at the bottom of the figure) is scanned and states within the layer are unranked and expanded, existing information on the solvability status of ranked successor states in the subsequent layer $n+1$ is retrieved.

Ranking and unranking with respect to the BDD is executed to look up the status (won/lost/drawn) of a node in the set of successors. We observed that there is a trade-off for evaluating immediate termination. There are two options, one is procedural by evaluating the goal condition directly on the explicit state, the other is a dictionary lookup by traversing the corresponding reward BDD. In the case of *Connect Four* the latter was not only more general but also faster. A third option would be to determine if there are any successors and set the rewards according to the current layer (as it is done in the pseudo-code).

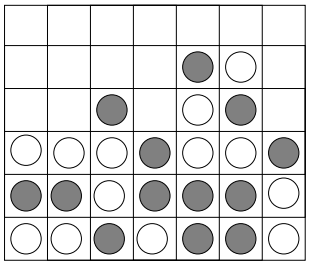


Figure 12.9: The game *Connect Four*: the player with the gray pieces has won.

To increase the exploration performance of the system we distributed the explicit-state solving algorithms on multiple CPU cores. We divide the bitvector for the layer to be solved into equally-sized chunks. The bitvector for the next layer is shared among all the threads.

For the ease of implementation, we duplicate the query BDDs for each individual core. This is unfortunate, as we only use concurrent read in the BDD for evaluating the perfect hash function, but the computation of the rank involves setting and reading local variables and requires significant changes in the BDD package to be organized lock-free.

12.9 Experiments BDD Hashing

Although most of the algorithms are applicable to most two-player games, our focus is on one particular case, namely the game *Connect Four* (see Figure 12.9). The game is played on a grid of c columns and r rows. In the classical setting we have $c = 7$ and $r = 6$. While the game is simple to follow and play, it can be rather challenging to win. This game is like *Tic-Tac-Toe*, with two main differences: The players must connect four of their pieces (horizontally, vertically, or diagonally) in order to win and gravity pulls the pieces always as far to the bottom of the chosen column as possible. The number of states for different settings of $c \times r$ is shown in Table 12.8.

Table 12.9 displays the exploration results of the search. The set of all 4,531,985,219,092 reachable states can be found within a few hours of computation, while explicit-state search took about 10,000 hours.

As illustrated in Table 12.10, of the 4,531,985,219,092 reachable states *only* 1,211,380,164,911 (about 26.72%) have been left unsolved in the layered BDD retrograde analysis. (More precisely, there are 1,265,297,048,241 states left unsolved by the algorithm, but the remaining set of 53,916,883,330 states in layer 30 is implied by the solvability status of the other states in the layer.)

Even while providing space in form of 192 GB of RAM, however, it was not possible to apply the symbolic solving algorithm to layers smaller than 30. The reason is, while the peak of the solution for the state sets has already been passed, the BDDs for representing the state sets are still growing.

This motivates looking at other options for memory-limited search and a hybrid approach that takes the symbolic information into account to eventually perform the complete solution of the problem.

12.10 Bibliographic Notes

Chess [98], *Checkers* [555] have shown competitiveness with human play. Pattern databases go back to [143] with *locality* having been studied by [677]. More general notions of locality have been developed by [363].

Table 12.8: Number of reachable states for *Connect Four*.

Layer	7×6	6×6	6×5	5×6	5×5
0	1	1	1	1	1
1	7	6	6	5	5
2	49	36	36	25	25
3	238	156	156	95	95
4	1,120	651	651	345	345
5	4,263	2,256	2,256	1,075	1,075
6	16,422	7,876	7,870	3,355	3,350
7	54,859	24,330	24,120	9,495	9,355
8	184,275	74,922	72,312	26,480	25,060
9	558,186	211,042	194,122	68,602	60,842
10	1,662,623	576,266	502,058	169,107	139,632
11	4,568,683	1,468,114	1,202,338	394,032	299,764
12	12,236,101	3,596,076	2,734,506	866,916	596,136
13	30,929,111	8,394,784	5,868,640	1,836,560	1,128,408
14	75,437,595	18,629,174	11,812,224	3,620,237	1,948,956
15	176,541,259	39,979,044	22,771,514	6,955,925	3,231,341
16	394,591,391	80,684,814	40,496,484	12,286,909	4,769,837
17	858,218,743	159,433,890	69,753,028	21,344,079	6,789,890
18	1,763,883,894	292,803,624	108,862,608	33,562,334	8,396,345
19	3,568,259,802	531,045,746	165,943,600	51,966,652	9,955,530
20	6,746,155,945	884,124,974	224,098,249	71,726,433	9,812,925
21	12,673,345,045	1,463,364,020	296,344,032	97,556,959	9,020,543
22	22,010,823,988	2,196,180,492	338,749,998	116,176,690	6,632,480
23	38,263,228,189	3,286,589,804	378,092,536	134,736,003	4,345,913
24	60,830,813,459	4,398,259,442	352,607,428	132,834,750	2,011,598
25	97,266,114,959	5,862,955,926	314,710,752	124,251,351	584,249
26	140,728,569,039	6,891,603,916	224,395,452	97,021,801	
27	205,289,508,055	8,034,014,154	149,076,078	70,647,088	
28	268,057,611,944	8,106,160,185	74,046,977	40,708,770	
29	352,626,845,666	7,994,700,764	30,162,078	19,932,896	
30	410,378,505,447	6,636,410,522	6,440,532	5,629,467	
31	479,206,477,733	5,261,162,538			
32	488,906,447,183	3,435,759,942			
33	496,636,890,702	2,095,299,732			
34	433,471,730,336	998,252,492			
35	370,947,887,723	401,230,354			
36	266,313,901,222	90,026,720			
37	183,615,682,381				
38	104,004,465,349				
39	55,156,010,773				
40	22,695,896,495				
41	7,811,825,938				
42	1,459,332,899				
Σ	4,531,985,219,092	69,212,342,175	2,818,972,642	1,044,334,437	69,763,700

Table 12.9: Number of Nnodes and states in (7×6) *Connect Four* (l layer, n BDD nodes, s states).

l	n	s
0	85	1
1	163	7
2	316	49
3	513	238
4	890	1,120
5	1,502	4,263
6	2,390	16,422
7	4,022	54,859
8	7,231	184,275
9	12,300	558,186
10	21,304	1,662,623
11	36,285	4,568,683
12	56,360	12,236,101
13	98,509	30,929,111
14	155,224	75,437,595
15	299,618	176,541,259
16	477,658	394,591,391
17	909,552	858,218,743
18	1,411,969	1,763,883,894
19	2,579,276	3,568,259,802
20	3,819,845	6,746,155,945
21	6,484,038	12,673,345,045

l	n	s
22	9,021,770	22,010,823,988
23	14,147,195	38,263,228,189
24	18,419,345	60,830,813,459
25	26,752,487	97,266,114,959
26	32,470,229	140,728,569,039
27	43,735,234	205,289,508,055
28	49,881,463	268,057,611,944
29	62,630,776	352,626,845,666
30	67,227,899	410,378,505,447
31	78,552,207	479,206,477,733
32	78,855,269	488,906,447,183
33	86,113,718	496,636,890,702
34	81,020,323	433,471,730,336
35	81,731,891	370,947,887,723
36	70,932,427	266,313,901,222
37	64,284,620	183,615,682,381
38	49,500,513	104,004,465,349
39	38,777,133	55,156,010,773
40	24,442,147	22,695,896,495
41	13,880,474	7,811,825,938
42	4,839,221	1,459,332,899
Total		4,531,985,219,092

Table 12.10: Result of symbolic retrograde analysis (excl. terminal goals, l layer, n BDD nodes, s states).

l	n (won)	s (won)	n (draw)	s (draw)	n (lost)	s (lost)
⋮	⋮	⋮	⋮	⋮	⋮	⋮
29	o.o.m.	o.o.m.	o.o.m.	o.o.m.	o.o.m.	o.o.m.
30	589,818,676	199,698,237,436	442,186,667	6,071,049,190	o.o.m.	o.o.m.
31	458,334,850	64,575,211,590	391,835,510	7,481,813,611	600,184,350	201,906,000,786
32	434,712,475	221,858,140,210	329,128,230	9,048,082,187	431,635,078	57,701,213,064
33	296,171,698	59,055,227,990	265,790,497	10,381,952,902	407,772,871	194,705,107,378
34	269,914,837	180,530,409,295	204,879,421	11,668,229,290	255,030,652	45,845,152,952
35	158,392,456	37,941,816,854	151,396,255	12,225,240,861	231,007,885	132,714,989,361
36	140,866,642	98,839,977,654	106,870,288	12,431,825,174	121,562,152	24,027,994,344
37	68,384,931	14,174,513,115	72,503,659	11,509,102,126	105,342,224	57,747,247,782
38	58,428,179	32,161,409,500	44,463,367	10,220,085,105	42,722,598	6,906,069,443
39	19,660,468	2,395,524,395	27,201,091	7,792,641,079	35,022,531	13,697,133,737
40	17,499,402	4,831,822,472	13,858,002	5,153,271,363	8,233,719	738,628,818
41	0	0	5,994,843	2,496,557,393	7,059,429	1,033,139,763
42	0	0	0	0	0	0

One early attempt to apply state space search on GPUs was made in the context of model checking [225, 66]. large-scale disk-based search has moved complex numerical operations to the graphic card [225]. As delayed elimination of duplicates is a performance bottleneck, parallel processing on the GPU was needed to improve sorting significantly. Since existing GPU sorting schemes did not show speedups on state vectors, refined GPU-based *Bucketsort* applies. In [66] algorithms for parallel probabilistic model checking on GPUs were proposed, exploiting the fact that probabilistic model checking relies on matrix vector multiplication. Since this kind of linear algebraic operation is implemented very efficiently on GPUs, these algorithms achieve considerable runtime improvements compared to their counterparts on standard architectures.

Cooperman and Finkelstein [135] have shown that two bits per state are sufficient to perform a breadth-first exploration of the search space. Efficient lexicographic ranking methods are studied in [61]. Many attempts, e.g. have a non-linear worst-case time complexity [424], and [419] employed lookup tables with a space requirement of $O(2^n \log n)$ bits to compute lexicographic ranks in linear time. Given that larger tables do not easily fit into SRAM, the algorithm does not work well on the GPU. Myrvold and Ruskey's algorithm [489] is linear in time and space for both ranking operations.

Two-bit breadth-first has first been applied to enumerate so-called *Cayley Graphs* [135]. Subsequently, an upper bound to solve every possible configuration of *Rubik's Cube* has been shown [427]: by performing a breadth-first search over subsets of configurations in 63 hours together, with the help of 128 processor cores and 7 TB of disk space it was shown that 26 moves always suffice to descramble it. Korf [418] has applied two-bit breadth-first search to generate the state spaces for hard instances of the *Pancake* problem I/O-efficiently. Peg Solitaire has been solved in [45], and an optimal player has been computed by [207]. Fore-and-Aft was originally an English invention, designed by an English sailor in the eighteenth century. Henry Ernest Dudeney discovered a solution of just 46 moves.

The breadth-first traversal in a bitvector repesentation of the search space was also essential for the construction of compressed pattern databases [81]. The observation that $\log 3$ are sufficient to represent all mod-3 values possible and the byte-wise packing was already made by [135].

Awari has been solved in 2002 [543] and the variant Oware strongly solved in 2021 [55] using bitboards and massive parallel search. For Chinese checkers even more refined ranking and processor instructions were used [600].

Perfect hash functions to efficiently (un)rank states have been very successful in traversing single-player problems [418], in two-player games [544], and for creating pattern databases [81]. The first reference to an ancestor of the game *Fox-and-Geese* is that of *Hala-Tafl* is believed to have been written in the fourteenth century. *Fox-and-Geese* is prototypical for cooperatively chasing an attacker. It has applications in computer security, where an intruder must be found. In a more general setting, such games are played with tokens on a graph.

Rubik's Cube, invented in the late 1970s by Erno Rubik, is a known challenge for single-agent search [414]. Sliding-tile puzzles have been considered in [350]. An external-memory algorithm that distributed states into buckets according to their blank position is due to [676]. The complete exploration of the 15-Puzzle is due to [419]. Top-Spin Puzzle has been studied in [117]. Nine-Men's-Morris boards have been found on many historic buildings; one of the oldest dates back to about 1400 BC [278]. Gassner has solved the game with endgame databases for the last two game stages together with alpha-beta search for the first phase [278]. Assuming optimal play of both players, he showed that the game ends in a draw. The pancake problem has been analyzed, e.g., by [176].

Botelho et al. [70] devise minimal practical hash functions for general state spaces, once the set of reachable states is known. BDDs [90] are very effective in the verification of hard- and software systems, where BDD traversal is referred to as *symbolic model checking* [91]. *Connect Four* has been weakly solved by [625, 13]. It has been shown that – while the reachable set leads to polynomially-sized BDDs – the symbolic representation of the termination criterion is exponential [206]. [160] have shown that the BDD ranking procedures work correctly. BDD perfect hashing [160] refers to ranking and unranking of states of a state set represented as a BDD. It is available in time linear to the length of the state vector (in binary). In other words, BDD ranking aims at the symbolic equivalent of constructing a perfect hash function in explicit-state space search [69].

Chapter 13
Card Game Playing

One central showcase of algorithmic intelligence is to prove that computers are able to beat humans in games. As many board games have either been solved or AIs show superhuman performance one of the next AI challenges are card games with randomness in the deal and incomplete information due to cards being hidden. While there is impressive research on playing multi-player non-cooperative card games like Poker, for cooperative card games like Skat or Bridge, humans were experienced to play better than computers.

Skat is an internationally played game, described by some as the best and most interesting card game for three players. It has a deck of 32 cards; in a deal each player gets 10 cards, with two left-over Skat cards. There are four stages of the Skat game: *i*) *bidding*, where the players communicate values towards their respective maximal bidding strength; *ii*) *Skat taking* and selecting the game; *iii*) choosing the two cards for *Skat putting*; *iv*) *trick-taking game play* with up to 10 rounds of play. To bid higher, stages *ii*) and *iii*) may be skipped (Hand). The winner of the bidding becomes the declarer, who plays against the remaining two opponents. He adds the Skat to his hand and discard any two cards. The declarer wins if he gets more than 60 points (Grand or Suit) or makes no tricks (Null). To increase the bidding value further, he can raise the contract from scoring 61 points to 90 (Schneider) and 120 (Schwarz), and to open the hand to the opponents (Ouvert). Handling partial information is the critical aspect in this game, given that for open cards the optimal score and the associated playing card can be found in terms of milliseconds.

We present a Skat AI, able to play all stages of the game and all game types. Using statistical tables elicited from Human expert games, it derives accurate winning probabilities, which are used mainly for the bidding and game selection stages, and to put good Skats. For the trick-taking stage of the game, it includes algorithmic strategies for opening, middle- and endgame play using expert rules and exploiting playing conventions to build a knowledge base on plausible and effective locations of the cards.

13.1 Introduction

Skat[1] is a three-player imperfect-information game played with 32 cards, a subset of the usual 52 cards Bridge deck. By the vast number of hands of possible deals, and an even larger number of different trick-taking play in card games, there is a limit of what expert game information can provide, and what search algorithms can uncover. While one can infer reliable statistical recommendations for the first cards to play, for later tricks in the middle game, this information is blurred, as with cards that have already been played, the lookup tables associated to card groups lose precision: e.g., it often makes a great difference if the highest or lowest card of the game is still in play or not.

To obtain dynamic information of the suit *factors* of the game we apply the following steps.

[1] This chapter is based on joint work with Samual Bounan and Rainer Gößl. It puts together and improves the work from [184, 185, 183, 186].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_13

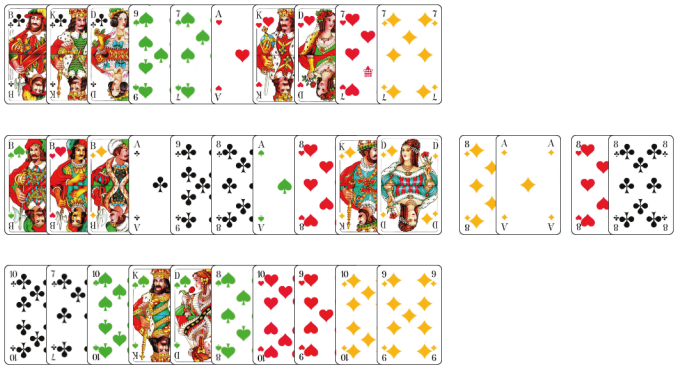


Figure 13.1: Skat game with player in the middle playing Grand.

1. Devise a single- and double-dummy minigame glassbox solver that efficiently determines the outcome of a game between the declarer and the opponent(s) restricted to one trump or non-trump suit, suggesting the card with the best possible payoff in optimal open card play.
2. Introduce to several applications of minigame solving for improving dynamic play and to determine the best possible card in a suit for the declarer and/or the opponent(s).
3. As the single-dummy algorithm becomes available in circumstances. where suits are not obeyed, the knowledge of the distribution of cards at least in some suits is maintained and updated accordingly.
4. Experimental findings based on playing a series of thousands of high-quality games show the impact of the minigame searches in improving the playing strength of an existing Skat AI.

Assessing the skill level of players to predict the outcome and to rank the players in a longer series of games is of critical importance for tournament play. Besides weaknesses, like an observed continuous inflation, through a steadily increasing playing body, the ELO ranking system, named after its creator Arpad Elo, has proven to be a reliable method for calculating the relative skill levels of players in two-player zero-sum games. The evaluation of player strength in multi-player trick-taking card games like Skat or Bridge, however, is not obvious. Firstly, these are incomplete information partially observable games with more than one player, where opponent strength should influence the scoring as it does in existing ELO systems. Secondly, they are game of both skill and chance, so that besides the playing strength the outcome of a game also depends on the deal. Last but not least, there are internationally established scoring systems, in which the players are used to be evaluated, and to which ELO should align. Based on the tournament scoring system, we suggested an ELO system for Skat to overcome these weaknesses.

Skat shares similarities to Marias(ch) (played in Czech Republic and Slovakia) and Ulti (played in Hungary). The rules of Skat go back to Hempel around 1848. Competitive Skat is defined by the International Skat Player Association. The game is played with three players. A full deck has 8 cards (A, T, K, Q, J, 9, 8, and 7) in all four suits (♣, ♠, ♡, ♢). After shuffling, each player receives 10 cards, while the skat consists of two cards. There are four phases of the Skat game: the bidding stage, taking and putting the skat, and the actual play for tricks. The *declarer*, who won the *bidding*, is playing for a win against the remaining two *opponents*. She is allowed to strengthen her hand by taking the skat and putting it (can be the same ones). For a strong Skat AI, especially for the early stages of the games like bidding, most interestingly is approximating the probability $P_w^t(h)$ of winning a given hand h in a game of type t. The games being played and bid for are *Trump*, which includes *grand* and *suit* (♣, ♠, ♡, or ♢), as well as *null*, a trick-avoiding variant of the game.

The bidding stage determines the declarer and the two opponents: two players announce and accept increasing bids until one passes. The winner of the first bidding phase continues bidding with the third player. The successful bidder of the second bidding phase plays against the other two. The maximum bid a player can announce depends on the type of game the player wants to play and, in case of a trump game, a multiplication factor

determined by the jacks. The soloist decides on the game to be played. Before declaring, he may pick up the Skat and then discards any two cards from his hand, face down. These cards count towards the declarer's score. An example for Skat selection is shown in Figure 13.1.

Card play proceeds as in Bridge, except that the trumps and card ranks are different. In grand, the four jacks are the only trumps. In suit, seven further cards of the selected suit are trumps. There are no trumps in null games. Non-trump cards are grouped into suits as in Bridge. Each card has an associated point value between 0 and 11, and in a standard trump game the declarer must score more points than the opponents to win. Null is an exception, where the declarer wins only if he scores no trick.

Skat has $n = \binom{32}{10}\binom{22}{10}\binom{12}{10} \approx 2.8$ quadrillion possible deals. Following the well-known *birthday paradox*, the probability p that at least one deal is repeated in k deals is $p = 1 - (\prod_{i=0}^{k-1}(n-i)/n^k)$; and fixing $p \geq 50\%$ yields $k \geq 40$ million games.

Nullspiel. There are four variants of the Nullspiel: *Null* (bidding value 23), *Null Hand* (35), *Null Ouvert* (NO, 46), and *NO Hand* (59), where *Ouvert* forces the declarer to show all her cards prior to the play, and *Hand* prohibits the declarer to take the skat. The declarer that wins the bidding, must lose all tricks. In this variant most computer card game programs play badly.

In the Nullspiel the ordering of cards is A, K, Q, J, T, 9, 8, and 7. If the declarer gets any trick, he loses. To the contrary, she wins by certain if her hand is *safe* (elaborate definitions include the cards in the skat and accommodate already played cards).

A declarer's card c is *safe*, if all gaps g in its suit with value lower than c are supported by at least the matching number of cards below g, with a special case for the declarer's turn in the first trick, where an extra support card is needed. The declarer's hand h is *safe* if all cards c in h are safe.

Strategies for the declarer and her opponents depend on the position of the players within the trick and dominate expert play. Playing agreements like *Shortest Suit – Smallest Card First* indicate the fundamental importance of collaboration between the two opponents for maximizing the exchange of information. Such hidden rules are difficult for an AI to learn automatically, especially given that for several of such simple rules, there are exceptions in world-class play.

There are other subtleties on knowledge elicitation given that there only two possible outcomes of the Nullspiel. One immediate consequence is that longer play often is a safer way to win the game of the opponents. In case a suit must be obeyed, the card distribution in this suit is crucial, and, if not, dropping card strategies play an important role.

For each of the suits s in a hand h we determine the probability of winning $P_w(h,s)$, using a multinomial distribution (refined with the winning probabilities found in the expert games). The probabilities are stored in tables, and the winning probabilities $P_w(s)$ among all the suits are multiplied as an estimate for the overall winning probability $P_w(h)$.

Trump Games. Card values in Trump games are added for the tricks made and the skat put, with a usual split at 60 of the 120 possible points. Other contracts (89, 119) for the declarer to exceed are possible. The bidding value depends on the distribution of Js: As the multiplier of the color value (12 = ♣, 11 = ♠, T = ♡, or 9 = ♢) or 24 (Grand), 1 is added to the number of consecutive Js in the order ♣, ♠, ♡, and ♢; or the number of consecutive Js in the joint hands of the opponents.

A declarer's non-trump card c is *standing* in a suit s, if c cannot be beaten by all other cards in s still being present in the game. The declarer's hand h is *standing* if all non-trump cards c in h are standing.

We implemented more refined definitions and classes of standing cards that depends on the distribution of the Js, the position in the trick, and that takes the possible distribution of opponent cards and the experience in expert games into account. To estimate the number of tricks possible, we, thus, introduce fractional values for standing cards and the concept of virtual standing cards that are likely to become standing cards during play.

13.2 Bidding

The usual bidding stage follows a predefined order of calling numbers corresponding to the *value* of the game $(18,20,22,23,24,27,30,33,35,\ldots)$. The final bidding value indicates in which suits the players are strong, so that we maintain a table of the at most two games fitting the final bid. In opponent play this information often determines the choice of the first card to be played.

For computer bidding we use a rather accurate knowledge on winning probabilities $P_w^t(h)$. In contrast to classical learning and game-theoretical approaches, we exploit flat statistical knowledge on the individual strength of the cards in hand that we have extracted from millions of expert games. We estimate probabilities $P_w^t(h)$ to win a game t and determine the expected payoff of the game t, which for the bidding stage is maximized.

Given $3\binom{32}{10,10,10,2} = 8{,}259{,}883{,}225{,}513{,}920$ different Skat deals (including the turn) storing a lookup table for $P_w^t(h)$ in memory clearly is infeasible and, even more importantly, for many million expert games, way too sparse to retrieve robust values for each game. Note the conceptual difference in P_w^t prior and posterior to skat taking. Before the skat is taken, the maximal bidding value is determined via computing the average payoff over all possible $\binom{22}{2} = 231$ skats. For putting we take the maximum over all $\binom{12}{2} = 66$ options to select the two cards for the skat.

The complete bidding strategy is more complex as the bidding value itself plays a role. Filter are applied on valid and good skats.

Once the skat has been taken by the declarer, there are $\binom{12}{2} = 66$ options for putting it. We derived a strategy to select a skat that optimizes the *dropping gain*.

The *dropping gain* is the change in the winning probability when removing a card from the hand. If h is the hand before the drop and h' is reduced hand after the card drop in suit s, then we have

$$drop(h,h') = P_w(h,s) - P_w(h',s).$$

In other words, for the choice of skat cards, we prefer the ones that improve the winning probability the most. For a better Skat putting strategy, refined winning parameters apply to grand and suit games. Moreover, strong players that announce the bid, often drop out one step before calling the actual value, not to be forced to play in case of a tie.

For the Nullspiel we follow the suggestion of Emanuel Lasker. and combine statistics on winning probabilities in each suit as follows $P_w(h) = \prod_{s\in\{\clubsuit,\spadesuit,\heartsuit,\diamondsuit\}} P_w(h,s)$. There are different probability tables $P_w(h,s)$ for other variants of the Nullspiel. In each suit, we have eight cards and $2^8 = 256$ selections of cards that form a pattern. As the binary representation of the position in the vector and the pattern are identical, given the bitvector for a given hand, we can extract the probability value $P_w(h,s)$ in time $O(1)$ by bit masking, bit shifting and bit reversal operations.

For Trump games we used hash functions to store similar hands in a smaller table (with several 10K entries). Such functions can be thought of selecting a set of features that generate equivalence classes of hands.

For example, in Grand games we identified the following winning features to be sufficient (in some cases, like bidding values and points put in skat, we cluster the set of possible values into a smaller set):

1. value put in the skat, we group the values into 5 classes;
2. trump quality encoding distribution in J: 10 classes;
3. number of As and Ts, as two independent features;
4. estimated number of lost cards based on standing ones;
5. position on table (who's turn it is);
6. bidding value, we group the values into 4 classes.

For Suit games, we identified nine features: trump count, number of trump high cards (As+Ts), encoding of Js, non-trump As and Ts, missing suits, position of player, encoded skat and bidding value. If we denote $f_1, \ldots, f_l$ for the l features, based on expert games we built a probability table $P_w(h) = hash(f_1(h), \ldots, f_l(h))$. Statistical evidence has been collected that the chosen features are indeed relevant. Choosing $P_w(h)$ for weakly supported feature combinations is a learning problem, which we resolve via returning a neighbor entry. We also tried to learn $P_w(h)$ with multivariable linear regression, but the predictions were often off by 20% and more.

There are many special rules to detect games won by certain. One general rule (high-card theorem) in Grand is the following. *If the number of high-value cards (HC) secured by the declarer is at least as large as the number of tricks lost (assuming no points were put into them), then the standard Grand game is won.*

As a proof of the statement, we look at the following cases.

1 Cached HC For one opponent trick, they certainly cannot reach 60 points.

2 Cached HCs For 2 tricks at most 44 points are available (4 As).

3 Cached HCs In 3 opponent tricks at most 58 points can be made (4 As, 1 T, and 1 K).

4 Cached HCs In 4 tricks at most 59 points are possible (3 As, 1 T, and 4 Ks).

5 Cached HCs In 5 lost tricks at most 57 points are contained (2 As, 1 T, 4 Ks and 3 Qs)

6 Cached Hs In 6 opponent tricks at most 54 points are possible (2 As, 1 T, 4 Ks, 3 Qs, and 2 Js)

7 Cached HCs In 7 opponent tricks we find at most 47 points (1 A, 4 Ks, 4 Qs and 4 Js)

If one has 4 As, then for the case of 4 cached HCs the declares can even afford giving 1 Q to the opponents, offering the opponents again at most 59 points in total.

The next concept to be understood is the concept of *standing cards*. Roughly speaking, a standing card is a (trump or non-trump) card, which will go home by certain. Standing cards are best viewed as sets of cards. We attach probabilities to them, which take, e.g,, the current turn, and the values put in the skat into account.

13.3 Open Game Play

In a search tree, it is often easy to cut some nodes that are unused, as in alpha-beta pruning algorithm . This kind of pruning is essential in MinMax tree because as shown by Knuth it reduces the search tree essentially to half of the depth.

For open card games we implemented an engineered solver to decide a game. The algorithm is a variant of $\alpha\beta$, but the game-theoretical value for trump games is found via binary search.

To represent hands, the skat and the played cards we employ bitvectors (in form of unsigned integers), for which we utilize Boolean set operations: `&` , $\sim$, and `|` . so that bit masking and shifts help to identify chosen parts of the hands in constant time. For an efficient solver, we exploit that modern CPUs provide constant-time `__builtin` procedures to determine the number of cards as `POPCOUNT(x)` (short `|x|`), the first card as `LZCOUNT(x)`, and the last card as `TZCOUNT(x)` (short `select`). For the representation of a *state* of the game, we chose the union of the hands.

Figure 13.2 provides the pseudo-code implementation for the Nullspiel, including transposition table pruning, detection of equivalent cards, and analyses of safe cards. We tried *proof-number search*, but with the involved handling of transpositions and the experienced higher efforts per node, it was less efficient. A concise implementation of the decision procedure to compute safe cards is given in Figure 13.3.

The backtracking algorithm in Figure 13.2 returns the game-theoretical value (lost, won) of the game at a given node in the search tree, with an And-node (AND) referring to the declarer and an Or-node referring to one of

Algorithm 13.1: $\alpha\beta(s,\alpha,\beta)$

```
αβ(s,α,β)
if over(s) then
  return score(s)
else if Min team then
  v ← +∞
  for all s' son of s do
    v ← min(v, αβ(s',α,β))
    if α ≥ v then
      return v
    β ← min(β,v)
else
  v ← −∞
  for all s' son of s do
    v ← max(v, αβ(s',α,β))
    if β ≤ v then
      return v
    α ← max(β,v)
return v
```

```
AND()
  if (v = lookup(hand[0]|hand[1]|hand[2],0)) return v;
  h = hand[0];
  while (h)
    index = select(h);
    bitindex = (1<<index);
    if (!playable(hand[0],index,0) || equiv(h,index))
      h &= ~bitindex;
      continue;
    hand[0] &= ~bitindex; played |= bitindex;
    t[0] = i[0]; t[1] = i[1]; t[2] = i[2]; i[0] = index;
    if (|played| % 3 == 0)
      turn = winner(0);
      i[0] = i[1] = i[2] = -1;
      if (turn == 0) val = 0;
      else if ((hand[0]|hand[1]|hand[2]) == 0) val = 1;
      else if (safe(hand[0],played) == hand[0]) val = 1;
      else if (turn == 1) val = OR1();
      else if (turn == 2) val = OR2();
    else val = OR1();
    i[1] = t[1]; i[2] = t[2]; i[0] = t[0];
    played &= ~bitindex; hand[0] |= bitindex;
    h &= ~bitindex;
    if (|played| % 3 == 0)
      add(hand[0]|hand[1]|hand[2],0,1);
    return 1;
  if (|played| % 3 == 0)
    add(hand[0]|hand[1]|hand[2],0,0);
  return 0;
```

Figure 13.2: Glassbox for the declarer's turn in the Nullspiel.

the two opponents (OR1 and OR2). Code fragments for the latter twos are similar, with the outcomes 0 and 1 reversed.

We see a lot of bitshifting to convert an index of a card to its position in the bitvector and vice versa. All variables not bound are global and set in a main driver program of the search. Variables set to some values are set back to their original one on a backtrack. We check for early termination in case the player only has safe cards. A transposition table (a hash table supporting insertion and membership only) avoids evaluating same game states again. We have two functions that check a card from a given hand h (maintained as a bitvector) is playable according to the rules of the game and not equivalent, meaning that a smaller card exists that will lead to the same game value, because these cards are adjacent. The recursive structure generates a tree and using Boolean

```
safe(hand, played)
  s = 0;
  for (suit=0;suit<4;suit++)
    counter = 0;
    for (j=7;j>=0;j--)
      card = 1 << (suit*8+j);
    if (card & played) continue;
    if (hand & card) counter++; s |= card;
    else if (counter == 0) break; counter--;
  return s;
```

Figure 13.3: Computing safe cards in the hand of the declarer with respect to a set of already played cards.

```
AND(remaining)
best = lookup(hands[0]|hands[1]|hands[2],as);
if (best != -1) return best;
while (remaining)
  index = select(remaining);
  bitindex = (1<<index);
  if (!playable(hands[0],index,0))
    remainng &= ~bitindex; continue;
  hands[0] &= ~bitindex; played |= bitindex;
  i[0] = index; val = -1;
  if (|played| % 3 == 0)
    cached = played; turn = winner(1,2,0);
    score = VALUE(i[0]) + VALUE(i[1]) + VALUE(i[2]);
    if (turn) gs += score; else as += score;
    t1 = i[1], t2 = i[2]; i[0] = i[1] = i[2] = -1;
    val =
     (gs >= 120-LIMIT) ? 0 : (as > LIMIT) ? 1 :
     (turn == 0) ? AND(hands[0]) :
     (turn == 1) ? OR1(hands[1]) : OR2(hands[2]);
    i[1] = t1; i[2] = t2;
    if (turn) gs -=score; else as -= score;
    cached = played;
  else
    val = OR1(playable(hands[1],(i[2]>=0)?i[2]:i[0]));
  i[0] = -1;
  played &= ~bitindex; hands[0] |= bitindex;
  remaining &= ~bitindex;
  if (val == 1)
    if (|played| % 3 == 0)
      add(hands[0]|hands[1]|hands[2],as+128);
    return 1;
  if (|played| % 3 == 0)
    add(hands[0]|hands[1]|hands[2],as);
  return 0;
```

Figure 13.4: Glassbox for the declarer's turn in Trump games.

reasoning to generate an optimal strategy. Pruning takes place in the Boolean formulas as only parts of it (e.g., the principal variation) need to be evaluated.

If three cards are on the table, the trick is evaluated and the game is either terminated or continued with the winner of the trick. To avoid problems with ongoing tricks, we use the transposition table only after a trick has been played. This is sufficient to encode the entire state into 32 bits, re-using the skat cards positions to denote the turn.

Figure 13.4 shows the according solver for Trump games, where point scores for both parties are added and hashed. To maintain the card order within the tricks, which is important to evaluate its outcome, we use explicit indices.

13.4 Skat Bot Architecture

Bidding and game selection both use statistical knowledge of winning ratios in human expert games, stored in tables and addressed via patterns of *winning features*. This assumes a predictor for a given hand with high accuracy before play (no move history). We assume the player's position to be part of his hand. The winning probability of a hand decreases during the bidding stage by the anticipated larger strength of the opponent hand. Other than this, no opponent model is used.

The Skat bot estimates winning probabilities with statistical tables that are extracted from a database of millions of high-quality expert games; more precisely, winning probabilities $Prob(h,p,s,b,t)$ including current hand h, choices of game type t, Skats s, position of the player p, and bidding value b. The probabilities are then used in the first three stages of the Skat game: bidding, game selection, and Skat putting. For each bidding value and each game type selected, it generates and filters all $\binom{22}{2} = 231$ possible Skats and takes the average of the payoff of skat putting, which, in turn, is the maximum of the $\binom{12}{2} = 66$ possible skats to be put.

The winning ratios in expert games can easily be analyzed statistically, but by the high number of $n = \binom{32}{10}\binom{22}{10}\binom{12}{10} \approx 2.8$ quadrillion possible Skat deals, proper generalizations are needed.

For Null games, given hand h and skat s the approach estimates the winning probability $Prob(h,s) = Prob(h,s,\clubsuit) \cdot Prob(h,s,\spadesuit) \cdot Prob(h,s,\heartsuit) \cdot Prob(h,s,\diamondsuit)$ addressed by the so-called winning parameters: number of non-trump suits that the player lacks; number of eyes in Skat, condensed into four groups; value of the bidding stage, projected to four groups; position of the declarer in the first trick; number of trump cards in hand; number of non-trump cards in hand; constellations of jacks condensed into groups; and number of cards estimated to lose, based on summing the expected number of standing cards.

Statistical tests showed that these parameters have a significant influence and can, therefore, be used as essential attributes to accurately assess the probability of winning a trump game. In particular, a *Grand* table with 113,066 entries is built on top of 7 of these winning parameters and a *Suit* table with 246,822 entries using 9 of them. For Skat putting we refine the lookup value for different cases in a linear function together with further winning features such as the expected number of tricks while respecting the retaking options of the issuing right, and the exact number of points put into the Skat.

Trick-taking is arranged wrt. an ensemble of different card recommendations. We find

- killer cards that force a win for the declarer (or the opponents) to meet (or to break) the contract of the game; this option mainly includes KBPS; other are simpler rules that count the number of points certain to be made for the player in the remaining tricks.
- endgame cards as the results of strategy fusion, realized via a voting on the winning ratio of open card game solver calls on the remaining *worlds* in the belief space of the player. The endgame player is invoked after five tricks with a maximum number of 2500 worlds in the belief space, the win ratio for a card (confidence level) is set to 90%. Additional bonus is given for a high number of eyes and for meeting higher contracts.
- hope cards as the only cards that can save the game for either the declarer or the opponents, i.e., all other cards lead to a forced loss, this card is played instantly
- expert cards for each player in each position in the trick, based on if-then-else rules that consider the current the hand of issuing players, the history of tricks being played, the partial knowledge of cards present in the opponent hands, etc.

The priority is as follows. First, killer cards are recommended; if this strategy fails to find a forced win, endgame and hope cards are searched for; if this does not meet the required criteria or confidence level, we fall back to expert cards recommendation. The expert rules, used for the first few tricks and as a default, includes card recommendations based on suit factors (either trump or non-trump). Each card in the factor is assigned a value 0, 1, or 2, where 0 denotes a hand card, 1 a card in the other players' hands, and 2 a card that is not playable (either being played or put into the Skat). For the declarer issuing trump we precomputed tables of sizes $\binom{11}{k} \cdot 2^{11-k} =$

11,264 ($k = 1$ trump), 28,160 (2 trumps), 42,240 (3 trumps), 42,240 (4 trumps), 29,568 (5 trumps), 14,784 (6 trumps), 52,80 (7 trumps), 1,320 (8 trumps), 220 (9 trumps), and 22 (10 trumps). For non-trump suits a table with $\sum_{k=1}^{7} \binom{7}{k} \cdot 2^{7-k} = 2,059$ entries is built.

13.5 Knowledge-based Paranoia Search

We represent the knowledge in the players as sets. To introduce the reasoning on the sets we give a brief example. Suppose we have the following deal

P_0 : ♡J, ♢J, ♡A, ♡K, ♡9, ♡7, ♣A, ♣8, ♣7, ♠A,
P_1 : ♣J, ♠J, ♡Q, ♣T, ♣K, ♣Q, ♠T, ♠7, ♢Q, ♢7,
P_2 : ♡T, ♡8, ♣9, ♠K, ♠Q, ♠9, ♠8, ♢A, ♢T, ♢8,
Skat: ♢K, ♢9

with the opponent P_2 to issue the first card. The game that is being played is ♡.

For P_2 we have the initial knowledge
$h_0 = h_1 = \{\}$, $h_2 = \{$♡T, ♡8, ♣9, ♠K, ♠Q, ♠9, ♠8, ♢A, ♢T, ♢8$\}$
$pool = \{$♣J, ♠J, ♡J, ♢J, ♡A, ♡K, ♡Q, ♡9, ♡7, ♣A, ♣T, ♣K, ♣Q, ♣8, ♣7, ♠A, ♠T, ♠7, ♢K, ♢Q, ♢9, ♢7$\}$
$skat = \{\}$, $declarerorskat = \{\}$, $partnerorskat = \{\}$
$noskat = \{$♣J, ♠J, ♡J, ♢J, ♡A, ♡T, ♡K, ♡Q, ♡9, ♡8, ♡7, ♣A, ♠A, ♢$A\}$

The declarer sees table card ♢A and updates his knowledge sets to $h_0 = \{$♡J, ♢J, ♡A, ♡K, ♡9, ♡7, ♣A, ♣8, ♣7, ♠$A\}$, $h_1 = h_2 = \{\}$, $skat = \{$♢K, ♢9$\}$
$pool = \{$♣J, ♠J, ♡Q, ♡T, ♡8, ♣T, ♣K, ♣Q, ♣9, ♠T, ♠K, ♠Q, ♠9, ♠8, ♠7, ♢T, ♢Q, ♢8, ♢7$\}$

The opponent player 1 now encounters ♢A, ♡A on the table and updates his knowledge sets to
$h_0 = \{\}$, $h_1 = \{$♣J, ♠J, ♡Q, ♣T, ♣K, ♣Q, ♠T, ♠7, ♢Q, ♢7$\}$, $h_2 = \{\}$, $skat = \{\}$
$pool = \{$♡J, ♢J, ♡K, ♡T, ♡9, ♡8, ♡7, ♣A, ♣9, ♣8, ♣7, ♠A, ♠K, ♠Q, ♠9, ♠8$\}$
$declarerorskat = \{\}$, $partnerorskat = \{$♢T, ♢K, ♢9, ♢8$\}$
$noskat = \{$♣J, ♠J, ♡J, ♢J, ♡T, ♡K, ♡Q, ♡9, ♡8, ♡7, ♣A, ♠$A\}$

To alleviate the computational burden, we introduce paranoia search to be initiated only after a few tricks have been played. The algorithm has been adapted to the knowledge already inferred by the Skat AI. It, thus, takes as an input *knowledge sets* corresponding to the inference that player P_i must have cards C_j (not) in his hand. This knowledge is inferred e.g., by unrealistically bad skats, players not obeying trump or non-trump cards and by playing conventions (putting the lowest-valued card in the declarer's trick, and the highest-value trick to the one of my partner, with some exceptions). As with many other parts of the Skat AI, for efficiency reasons, sets of cards are encoded as bitvectors of length 32 (unsigned int). This allows fast bit manipulation, such as card selection and copying. The minimax alpha-beta simulating *moving-test-driver* search algorithm trump games is implemented as a binary search (see Figure 13.5) over an AND/OR tree decision procedure (see Figure 13.6) that returns, whether or not the declarer can win the game according to a given contract limit.

It progresses belief sets for partial information. Knowledge-based paranoia search (KBPS) can be applied in forehand, in middlehand, and in rearhand positions of the players. Besides updating knowledge vectors, scoring values, current contract limit, the call must respect played cards on the table to trigger a correct analysis. By monitoring server logs online during play, we validated the working of the algorithm: once a win has been found it persists to the end, in many cases long before the human opponent recognized that he is lost.

Paranoia Search for the Declarer. The worst-case search option for the declarer is used in trump games. Its implementation as a moving test driver is a binary search loop over a backtracking branch-and-bound procedure

```
solve(hand0,hand1,hand2,doublehand,played,as,gs,table)
  left = as-1, right = 120
  while (true)
    if (left == right-1) return right
    limit = (left+right)/2
    x = run(hand0,hand1,hand2,doublehand,played,as,gs,table)
    if (!x) right = limit else left = limit
```

Figure 13.5: Moving test driver for declarer knowledge-based paranoia search; hand0 are the cards of the declarer, hand1/hands2 opponent cards known to him, doublehand is the pool; unknown, which hands the cards belong.

```
run(hand0,hand1,hand2,doublehand,p,as,gs,table)
  h[0] = hand0; h[1] = hand[1]; h[2] = hand2, i = table;
  aspts = as; gspts = gs, pool = doublehand; played = p
  return AND(hand0);
```

Figure 13.6: Running declarer knowledge-based paranoia search; starting and or tree search after set some global backtracking variables.

to find the optimal game value. As we use the search option dynamically, the algorithm is initiated after a fixed number k of played cards. In the overall architecture it acts as a prioritized killer card proposal that warrants a forced win. Paranoia search takes the partially played game, and a set of possible worlds as a parameter, encoded as knowledge sets, and contract bound. We limit the uncertain knowledge to the sets of *free* cards that are still to be distributed among the two opponent players. All *fixed* cards are assigned to one hand. Figure 13.7 shows the implementation of the declarers' KBPS backtracking algorithm at an OR node for the first opponent in the AND-OR partial observable search tree. It determines if a game can be won against all worlds according to a given score bound *limit* as fixed by the overall binary search. For the sake of simplicity, we omit code for transposition table pruning cards.

As the declarer knows that Skat (except for *Hand* games), as with the above overall knowledge representation and reasoning example there are 3 knowledge sets provided to the player: *pool*, denoting all remaining cards not yet known on which opponent hand they reside, h_1, cards already known to be in the 1st opponent hand, h_2, cards already known to be in the 2nd opponent hand. Furthermore, we have *avail*: hand cards playable according to the rules of Skat, obeying trump and suit; *index*, *bit*: selected card, for being played; *played*: cards already played; w: winner of trick; i_0, i_1, i_2: table cards by players; r_1, r_2 number remaining cards available; *limit*: current bound for game value; *score*: card value of table cards; *aspts*: point total for the declarer (according to the given knowledge of the Skat); *gspts*; point total for the opponents.

The KBPS algorithm searches the tree of playable cards, and branches wrt. the set of known cards and the current belief, while respecting the rules of play and the number of cards that a player can have. If suits are not obeyed, knowledge vectors for cards available to each hand are updated during the search. Before cards are selected from the pool of cards available to both players, they are assigned to one opponents' hand.

The algorithm can be extended to cover more knowledge inference options like playing conventions for the opponents such as giving the highest-valued card to a trick that goes to the partner, and a lower-valued card to a trick the declarer.

If the capacity of a hand is exceeded, we encounter a dead-end and a backtrack is initiated. In other words, if more cards are assigned to the player than his hand can hold, the entire subtree is pruned. By the virtue of enumeration of all card combinations, the algorithm computes the game-theoretical partial information (minimax) score, assuming optimal play of the players. The transposition table and equivalent card pruning are implemented in a way not to violate this outcome.

```
OR1(avail)
  avail = playable(avail,h[1],i)
  while (avail)
    index = select(avail);
    bit = (1<<index);
    h2 = h[2]; o = pool
    if (c = first-card-on-table(i))
      if (trump & (1 << c))
        if (|trump & bit| == 0)
          h[2] |= trump & pool;
          if (exceeded(h[2]))
            h[2] = h2;
            avail &= ~bit;
            continue;
          pool &= ~h[2];
      else
        if (|suit(c) & bit| == 0)
          h[2] |= suit(c) & pool;
          if (exceeded(h[2]))
            h[2] = h2;
            avail &= ~bit;
            continue;
          pool &= ~h[2];
    if (exceeded(h[1]|bit))
      (h[2],pool) = (h2,f);
      avail &= ~bit;
      continue;
    (h1,p,i[1],r) = (h[1],played,index,-1);
    pool &= ~bit; h[1] &= ~bit; played |= bit;
    if (endoftrick(i))
      w = winner(2,0,1);
      score = value(i);
      gspts += w ? score ; 0;
      aspts += w ? 0 : score;
      (i0,i2) = (i[0],i[2]); i[0] = i[1] = i[2] = -1;
      r = (gspts >= 120-limit) ? 0 :
          (aspts > limit) ? 1 :
          (w == 0) ? AND(h[0]) :
          (w == 1) ? OR1((pool|h[1]) & ~h[2]) :
          OR2((pool|h[2]) & ~h[1]);
      i[0] = i0; i[2] = i2;
      gspts -= w ? score : 0;
      aspts -= w ? 0 : score;
      (i[1],h[1],h[2],pool,played) = (-1,h1,h2,o,p);
    avail &= ~bit;
    if (r == 0) return 0;
  return 1;
```

Figure 13.7: Declarer KBPS at an OR search node for the first opponent's selection of a card; other search node implementations are simpler or similar.

A proof that a win is forced and will not be lost during subsequent play can be done by induction on the number of remaining cards to be played, but is obvious, as the game-theoretical minimax value is computed at the root node. If the knowledge is exact, i.e., given that no false information is contained in the knowledge vectors, then the algorithm progressing the vector does not falsify it. We are claiming soundness but not correctness, i.e., if no forced win is found, then the game continues with other card recommendations.

Given that the knowledge provided in the knowledge vectors is valid at invocation time of the algorithm, once value 1 is returned by the KBPS declarer algorithm (cf. Figure 13.7), the game is won by the declarer and this forced win will manifest during subsequent trick-taking play. If the algorithm optimizes the number of points in the moving test driver, the declarer will receive more points than the computed limit.

Paranoia Search for Schneider & Schwarz. When a game can be won to the contract of 61 points, it is desirable to aim at Schneider (90 pts) or Schwarz (120 pts). This is done by restarting the analysis with a higher contract, once the one for the current limit has been proven to be a win.

Approximate Paranoia Search. The worst-case analysis has two major limitations. As stated in Theorem 1, the AIs act *in paranoia*. Suppose that all non-trump cards of a suit are neither in the declarers hand nor in the Skat, then even extreme distributions of the cards with all cards on either hand have to be accounted for in the analysis. The probability for this case, however, is only 1.5625%. The virtue of good Skat play is to play well against most likely and not all card distributions. For the approximate KBPS algorithm, we, therefore, demand that certain distributions of cards are unlikely, and should be excluded from the search. Secondly, the running time is larger in case of more uncertainty, so that belief space measured in the number of worlds the AI plays against, may hinder finishing a complete KBPS exploration in time.

Both objectives can be met together by limiting the cards that can be assigned to each hand. This is the basis of *approximate knowledge-based paranoia search* (AKBPS), that poses constraints on the cards distributions allowed on each hand, or —for implementation purposes— enforces some cards assigned to a hand. Of course, the theorem no longer holds, as there are some worlds that are not considered, still the observation is, that early suggestions of cards that wins against all, but extreme worlds are extremely valuable. In contrast, Furtak used lower bounds, and set the declarer cut-off to 57.

Paranoia Search for the Opponents. Extending the approach from the declarers' point of view to the ones of the opponents is tricky, mainly due to the presence of the unknown Skat. For example, if one of the other players does not obey, it is no longer immediate that the card is on the remaining player's hand, as it can reside in the Skat. In the knowledge-based paranoia search algorithm, illustrated for the case of the declarer's AND node in Figure 13.8 (for efficiency reasons, we are using many bitvector set operations!), this leads to the introduction of further knowledge vectors. We now have five sets that are updated denoting that the declarer or the Skat has a card, or that the opponent, or the Skat has a card of the pool of remaining cards, only if taken or a card is known to be on the hand, e.g., by selecting it, it is moved. In some respects, the knowledge sets (*declarerorskat* and *partnerorskat*) are caches for the main pool of cards (*pool*) for the remaining players. In some of the conditions applied we take care that no more cards are moved to a hand than it can cope with.

If one opponent sees a definite win, this does not mean that the other opponent sees it as well. Given a different set of hand cards he may have very different knowledge on the distribution of cards. As it is defined, it requires one defender to assume that his partner will intentionally play poorly. Again, soundness can be proven by induction of the remaining cards to be played, and the observation that a search tree with less remaining cards is part of a search tree with more remaining cards, leading to a forced win. According to the uncertainty in the Skat there are three pools of cards that reflect the rising knowledge instead of one.

Given that the knowledge provided in the knowledge vectors is valid at invocation time of the algorithm, once a card is returned by the KBPS opponent algorithm (cf. Figure 13.8), the game is won by the opponent and this forced win will manifest during subsequent trick-taking play. If the algorithm optimizes the number of points in the moving test driver, the declarer will not receive more points than the computed limit.

Worst-Case Analysis for Avoiding Schneider/Schwarz. In opponent play, using a paranoid assumption on the card play is less effective than for the declarer play, and often applies to the endgame analysis. When the game is won by the declarer, however, KBPS, frequently applies to avoid a high loss with 90 declarer points, called Schneider, or a maximum loss with 120 declarer points. Therefore, once the contract of the declarer has been achieved, we use KBPS in opponent play with a scoring limit for Schneider/Schwarz.

```
AND(avail)
  while (avail)
    index = select(avail);
    bit = (1<<index);
    (h0,h2,o,as,ms) =
        (h[0],h[2],pool,declarerorskat,partnerorskat);
    if (c = first-card-on-table(i))
      if (trump & (1 << c))
        if (|trump & bit| == 0)
          partnerorskat |= trump & pool;
          if (exceeded(partnerorskat))
            h[2] = h2; partnerorskat = ms;
            avail &= ~bit;
            continue;
          pool &= ~partnerorskat;
          h[2] |= noskat & partnerorskat;
          if (exceeded(h[2]))
            (h[2],pool,partnerorskat) = (h2,o,ms);
            avail &= ~bit;
            continue;
          partnerorskat &= ~h[2];
      else
        if (|suit(c) & bit| == 0)
          partnerorskatat |= suit(c) & o;
          if (exceeded(partnerorskat))
            (h[2],t,partnerorskat) = (h2,o,ms);
            avail &= ~bit;
            continue;
          pool &= ~partnerorskat;
          h[2] |= noskat & partnerorskat;
          if (exceeded(h[2]))
            (h[2],pool,partnerorskat) = (h2,o,ms);
            avail &= ~bit;
            continue;
          partnerorskat &= ~h[2];
    if (exceeded(h[0]|bit))
     (h[2],partnerorskat,pool) = (h2,ms,o);
     avail &= ~bit;
     continue;
    p = played; pool &= ~bit;
    h[0] &= ~bit;h[2] &= ~bit;
    declarerorskat &= ~bit;
    partnerorskat &= ~bit;
    played |= bit; i[0] = index; r = -1;
    if (endoftrick)
      w = winner(1,2,0);
      score = value(i);
      ap = aspts; gp = gspts;
      gspts += w ? score: 0;
      aspts += !w ? score: 0;
      t1 = i[1], t2 = i[2];
      if (|played| == 30) aspts += value(~played);
      i[0] = i[1] = i[2] = -1;
      r = (gspts >= 120-limit) ? 0:
          (aspts > limit) ? 1:
          (w==0)? AND((pool|declarerorskat|h[0]) & ~h[2]):
          (w==1)? OR1(h[1]):
          OR2((o|partnerorskat|h[2]) & ~h[0]);
      i[1] = t1; i[2] = t2;
      gspts = gp; aspts = ap;
    else r = OR1(feasible(h[1],i));
    (h[0],h[2],played,declarerorskat,partnerorskat,pool) =
      (h0,h2,p,as,ms,o);
    avail &= ~bit; i[0] = -1;
    if (r == 1) return 1;
  return 0;
```

Figure 13.8: Opponent KBPS at an AND search node for a declarer's selection of a card; other search node implementations are simpler or similar.

13.6 Factorized Search

Factorizing the search is a concept well known from combinatorial game theory [45] and has been applied to endgame play in Computer Go [487]. The main approach, readily applicable to simple games such as Nim and put forward to a divide-and-conquer search strategy is to decompose the large game into smaller one, whose combination can exactly solve the original games. We aim much lower, and look at mini searches in the factors for each of the suits (either trump or non-trump) to approximately solve some of the problems in dynamically optimizing play; providing card suggestions for the case, when information about card become clarified, e.g. due to non-obeying suits.

The so-called minigame solver is widely usable to bridge the opening of the game, where we can apply reliable statistical information for proper card recommendations, to the endgame, where we can analyze the entire belief-space. We give some examples in the game of Skat, but the approach applies to many other trick-taking card games as well. In particular, minigame search can be used, e.g,

- for book-keeping trumps in opponent play, where we have to select a trump card and evaluate the strength of the play against the declarer. Suppose ♣ is trump, that the declarer issues ♡J, and is known to have ♣J,♣D,♣9,♣8 as remaining card in hand, with one opponent bailing out on trump. The other opponent has ♢B,♣A,♣T, so that only looking on trump 28 points can be made. If we further assume that the declarer gets back the right to open a trick. Minigame search determines that assuming optimal play that 17 points are available for the declarer, suggesting to playing card ♣T first, on ♣J playing ♢J second, holding back the ♣A for the 11 points difference.
- For book-keeping non-trumps for the opponents. If we were to know that the declarer has ♢A,♢K,♢9, and ♢7 in one suit and the others three of the total seven cards are located at one opponent, then he knows that he has to overtake an issued no-value card played by the declarer with the Q to save the ♢10. Similarly, for ♢10, ♢K, ♢9 in the declarers hand the opponents will be able to sack the Ten if the Ace is standing with two other cards.
- for the dynamic analysis of standing cards for an estimate on the number of tricks in a suit that will go home eventually. The first card to issue can be extracted from expert card database, but in the middle game, this lacks dynamic information about trick-taking. The number of tricks (if any) to be secured by the opponents can be counted and indicate, which card the declarer must show. Based on this information in the endgame it is even possible to detect so-called non-trump or trump *forks*.

13.6.1 The Power of Suit Partitioning

For variants of the Null-Games (Null, Null-Hand, Null-Ouvert, Null-Ouvert-Hand, the misère game variants in Skat), approximating the winning probabilities $P_w(h)$ of a hand h multiplying the projected ones $P_w(h \downarrow s)$ in each suit, $s \in \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$ is astonishingly accurate, i.e,

$$P_w(h) \approx \prod_{s \in \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}} P_w(h \downarrow s).$$

If h is represented as a set of cards and s is represented by the set of all cards in a suit, then $P_w(h \downarrow s) = P_w(h \cap s)$.

When we use bitvectors for sets and their manipulations concept of patterns in each suit can be enumerated, for a full suit of eight cards in the Null game the *card group* $CG_8(h,s) = h \downarrow s \in \{0,\ldots,255\}$. For non-trump we have card groups $CG_7(h,s)$ and for trump play either $CG_{11}(h,s)$ (Suit), or CG_4 (Grand). It is immanent that for GG_k there are at most 2^k hands. Given that there are three different players and cards that already been played, pre-computing all possible card combination for a fast lookup in a hash table is cumbersome and leads to very large tables.

In the static variant, where the groups have not been touched within the tricks being played, we generate tables of size 2^8 with the probabilities for winning have to be pre-computed using the large amount of stem games. The concept of such table is reappearing for many different aspects of the Skat game, such as color change, first cards to issue, cutting cards in suit games, etc.

Unfortunately, the statistical information does not easily cover the dynamics of on-going games, as the CGs are affected by the removal of cards being played. As an example of the concept of dynamic suit partitioning is the computation of safe cards, since a hand of the declarer in a Null game is safe (100% certain), if it is safe in each of the suits.

For trump games (Grand/Suits) the concept of accumulating static information in each suit (either trump or non-trump cards only) is also reflected in some of the features of a hand, to determine the winning probability. One such feature is the number of trump aces and non-trump aces and tens, or free suits, which are empty groups. A critical one is estimating the number of *standing cards* in each suit, a player expects to win. The approximate values for the standing cards are added for the entire hand,

$$standing(h) = \sum_{s \in \{\clubsuit, \heartsuit, \spadesuit, \diamondsuit\}} standing(h \downarrow s),$$

where $standing(h \downarrow s) = standing(h \cap s)$. For estimating $standing(h \downarrow s)$ statistical information is collected.

While this works well for bidding, for Skat putting and for the early stages of trick play, the dynamic evolution of KGs, e.g., for standing cards, proves to be a real challenge. Note that the truely dynamic algorithm for safe suits and hands that considers the cards being played is not easily to be extended, as in trump games the right to issue cards often changes, and we want to count, tricks and points.

Therefore, a dynamic concept of standing cards is needed, which we call *minigames*, i.e., the reduction of the overall game to one suit or the set of trump cards, aggregating the values for the entire game, possible averaging with respect to the amount of uncertainty in the cards. In minigames, only non-trump cards of one suit, or trump cards are issued. To overcome the problem we use an open card mingame solver.

13.6.2 Miniglassbox Search

Open or dummy card solving is a glassbox search with perfect information. There are single-dummy and double-dummy variants of the mini glassbox search algorithm, MGSDS and MGDDS for short. These algorithms compute the best playing card wrt. the optimization objective the trick starting condition, while assuming optimal play of the opponents. The search spaces are so small that we omit transposition table pruning [532], as exploration in these state spaces is negligible and this would induce re-initializing the hash table after each run.

The exploration algorithm for such open card solvers, is decomposed into decision procedure on a given threshold for the point and embedded in a binary search for computing the optimum (a structure also known as *moving-test driver*.

13.6.3 Single-Dummy Miniglassbox Solver

The simplest case is given if one of the three players is known to not have a card of a certain (non-trump or trump) suit s, e.g., by not obeying s. Then the minigame reduces to only two parties that remain to hold cards of s in their hands. Except of the cards put in the Skat, full knowledge of all cards in s is known. There are also other measures monitoring the play to deduce if one player cannot have any card of a certain kind, as one may generally assume that the lowest-value card is given to the the trick owned by the opposing party, and the highest-value card to the trick owned by the own party.

```
MiniSingle::AND(playable)
if (!playable)
  return opoints + sum(h[1]) < maximum - limit;
while (playable)
  index = select(playable); bitindex = (1<<index);
  h[0] &= ~bitindex; played |= bitindex;
  t[0] = index; rval = -1;
  if (|played| mod 2 == 0)
    turn = winner(1,0);
    score = VALUE(t[0]) + VALUE(t[1]);
    if (turn) opoints += score; else dpoints += score;
    t1 = t[1]; t[0] = t[1] = -1;
    rval = (opoints >= maximum-limit) ? 0
    : (dpoints > limit) ? 1 : AND(h[0]);
    t[1] = t1;
    if (turn) opoints -=score; else dpoints -= score;
  else
    rval = OR(h[1]);
  t[0] = -1;
  played &= ~bitindex; h[0] |= bitindex;
  playable &= ~bitindex;
  if (rval == 1) return 1;
return 0;
```

Figure 13.9: Mini single-dummy miniglassbox solver for declarer node, counting points.

In practice the single-dummy variant of the minigame solver applies to trump play in suit games in order to decide on the trump card for an opponent to obey, or to a player to decide whether or not to start issuing trump cards from top to bottom. At this stage of the game, it has been clarified that the other player has no trump card left.

The algorithm counts the number of points available for the declarer, and assumes optimal play of both parties,

Notice a few subtle insights. While the result of the trick is correctly distributed to the players the next turn will always be at the declarer, which compensates that he tries to win and is assumed to get back to play with his stronger by-hand.

The reasoning is that the declarer is strong and will get its right to issue the card. The single-dummy algorithm is given in Figure 13.9, so that there are two players, of which we selected the declarer node. The algorithm is tuned for speed. It uses bitvectors for the hands, suitmasks, olayed and playable cards. Function select extracts one bit in the set (via constant-time processor instructions), winner determines the leader in a trick. Hand cards are $h[j]$, table cards are $t[j]$, with j being Player $j \in \{0,1,2\}$. If the declarer (Player 0) runs short of playable trumps, all remaining cards are counted for the opponents (opoints). Otherwise, a card is selected.

There are some subtle issues to be solved on the first trick, as some cards may be already present on the table. They also influence the maximum of points that can be reached. As we count points, we take into account the value of the partner card in the first trick.

13.6.4 Double-Dummy Miniglassbox Solver

The double-dummy version of the miniglassbox solver is shown in Figure 13.10, also for the declarer's turn. Besides being defined for three players, it aligns with the notation of variables in the single-dummy version of Figure 13.9.

The double-dummy miniglassbox solver also shares similarity with a double-dummy solver for the overall game. But the search space is much smaller, as the game is restricted to only one suit. Again, we skip the transposition

```
MiniDouble::AND(playable)
if (!playable) return dpoints > limit;
while (playable)
  index = select(playable); bitindex = (1<<index);
  h[0] &= ~bitindex; played |= bitindex;
  t[0] = index; rval = -1;
  if (|played| mod 3 == 0)
    turn = winner(1,2,0);
    score = 1;
    if (turn) opoints += score; else dpoints += score;
    t1 = t[1], t2 = t[2];
    t[0] = t[1] = t[2] = -1;
    rval =
      (opoints >= maximum-limit)? 0:
      (dpoints > limit)? 1:
      (turn==1 && h[0] && h[1] & suit)? OR1(h[1]):
      (turn==2 && h[0] && h[2] & suit)? OR2(h[2]):
      AND(h[0]);
    t[1] = t1; t[2] = t2;
    if (turn) opoints -=score; else dpoints -= score;
  else
    rval = OR1(playable_cards(h[1]);
  t[0] = -1;
  played &= ~bitindex; h[0] |= bitindex;
  playable &= ~bitindex;
  if (rval == 1) return 1;
return 0;
```

Figure 13.10: Mini double-dummy miniglassbox solver for declarer node, counting tricks.

table pruning. We use it for counting the number of tricks, not for not counting points, but this can easily be changed. As we see there are now three players AND, OR1, and OR2. first trick.

In this case we want the number of tricks as a measure about to generate standing non-trump cards for the declarer. Again the decision variant for a given threshold is shown.

An example for the algorithm for building standing cards is as follows. Assume the declarer has A, K, 8, 7 in some arbitrary suit, and plays the 8 according to the static table.

Case 1. Both opponents obey the suit, then only one card is in the opponents' hands, and the ace is the highest card, so that the declarer gets all tricks if he plays from above, as he has three standing cards.

Case 2. One opponent does not have a remaining card in the requested suit. If the 10 drops immediately, the AS wins all further tricks, given that he issues from highest- to lowest-value card. If one opponent overtakes the trick with the Q, the declarer has to take the Ace to lose the 7. For this case he only has 2 standing cards.

For the glassbox solver we impose that the declarer gets the right to issue the next card after a trick has been cached and counted for the correct party, only if there is no other card of the suit to be played.

The turn (which player has to select the next card) for the next trick is only approximated. Setting it only to the declarer neglects finessing the cards, which is an important factor in almost all trick-taking games.

The binary search optimization algorithm for the exact value assuming optimal play is called for each possible card distribution of the opponents. As this distribution might be uneven, to allow playing on, we pad cards of different suits to the opponents.

The number of cards in a suit follow a hyper-geometric distribution [300]. In Figure 13.11 we illustrate how to generate all possible remaining opponent hands, given the hand for the declarer. The algorithm is recursive and puts all possible cards still to distribute in either the one or the other hand until no opponent card remains. The result of evaluating all calls with the minigame glassbox solver is added, and later averaged for the number of possible distributions.

```
MiniDouble::solve(h0,h1,h2,play,P0,P1,P2,start)
  left = -1, right = maximum;
  while (true)
  if (left == right-1)
    return right;
  limit = (left+right)/2;
  x = MGBS(h0,h1,h2,play,P0,P1,P2,start);
  if (!x) right = limit; else left = limit;

MiniDouble::find(h0,h1,h2,first) {
  pad1 = count(hand0) - count(h1) + (first != -1);
  pad2 = count(h0) - count(h2) + (first != -1);
  h1 = pad(h1,pad1);
  h2 = pad(h2,pad2);
  maximum = count(h0) + (first != -1);
  if (first == -1)
    return solve(h0,h1,h2,0,-1,-1,-1,0);
  else
    return solve(h0,h1,h2,(1<<first), first,-1,-1,1);
  return v;

  MiniDouble::distribute(decl,opps,o0,o1,first)
  if (opps == 0)
    return find(declarer,h0,h1,first);
  index = select(opps); bit = (1<<index);
  aver1 =
    distribute(decl,opps & ~bit,o0 | bit, o1,first);
  aver2 =
    distribute(declarer,opps & ~bit,o0, o1 | bit,first);
  return aver1 + aver2;
```

Figure 13.11: Calling the Mini double-dummy glassbox solver for all possible distributions of opponent cards.

13.7 World Search

The knowledge is complex, it includes simpler aspects like cards being played, the eyes of tricks taken by the players, the cards being played but also if the player is certain whether a certain card is at the other player's hand or in the skat. It is updated with each card that is played.

Let $\mathscr{P} = \{p_0, p_1, p_2\}$ be the players (p_0 is the declarer, p_1 and p_2 are the opponents). Let $\mathscr{C} = \{c_0, \ldots, c_{31}\} = \{\clubsuit A, \ldots, \diamondsuit 7\}$ be the set of all cards, and $\mathscr{B}_p$ be the belief space at the current moment of play with player p to move. Let the proposition $legal(p,s,c)$ denote that a card $c \in \mathscr{C}$ is playable for player p in state $s \in \mathscr{B}_p$, proposition $knows(p,s)$ denote that a state $s \in \mathscr{B}_p$ is consistent with the current knowledge of p, and $\mathscr{K}_p = \{s \in \mathscr{B}_p \mid knows(p,s)\}$ be the set of consistent states.

Random deals and hidden cards result in incomplete information and partial observations. In incomplete information search like in card games, we usually have many different *worlds* (alias states) in the belief space of the players. The number of possible worlds goes down with each card that is played and with each bit of information extracted from the tricks. There are two main search options for incomplete-information games, which —depending on the length of the play— are applicable mainly to endgame play. The core difference for two approaches for card selections put into quantifiers are as follows.

Open-Card Solvers: All Worlds – One Card $\forall_{s \in \mathscr{K}_p} \exists_{c \in \mathscr{C}} legal(p,s,c) \wedge win(p,s,c)$ In this case, each individual world is evaluated, and different cards for different worlds may be proposed, so that there needs to be some fusion like a voting. In several cases the entire belief space is sampled, sometimes weighted with the probability of a card in a hand.

World Search: One Card – All Worlds $\exists_{c \in \mathscr{C}} \forall_{s \in \mathscr{K}_p} legal(p,s,c) \Rightarrow win(p,s,c)$ This requires a play against all worlds in one search tree, branching on the location and selection of cards.

Both searches have their pros and cons, in term of the quality of the card recommendation, and the performance to calculate it, so that we have included both in the player.

13.8 Open-Card Solver Voting

Voting for Null games is a simple majority on the games that are won by the player to select a card. For trump games, we apply a slightly more complex voting, and let $score(s,c)$ be the game value for the score of the declarer in the open-card solver given s, $declarer\text{-}eyes(s)$ the eyes collected by the declarer so far, and c the recommended cards that meet the optimal score. We selected the following voting scheme.

1. For $c \in \mathscr{C}$ in s initialize $votes(c)$ with 0.
2. For each state s in the belief space $\mathscr{K}_p$ and legal card $c \in \mathscr{C}$ to be played in s:

   opponent card Set $votes(c)$ by $120 - score(s,c)$, and
   - if $(score(s,c) \leq 60)$ then $votes(c) \leftarrow votes(c) + 10$.
   - if $(score(s,c) < 90)$ then $votes(c) \leftarrow votes(c) + 5$.
   - if $(declarer\text{-}eyes > 60 \wedge score(h,c) < 90)$ then $votes(c) \leftarrow votes(c) + 5$.

   declarer card increase $votes(c)$ by $score(s,c)$, and
   - if $(score(s,c) > 60)$ then $votes(c) \leftarrow votes(c) + 10$.
   - if $(score(s,c) \geq 90)$ then $votes(c) \leftarrow votes(c) + 20$.

Let $z = \Sigma_{c \in \mathscr{C}}\ votes(c)$ and α some confidence threshold (e.g, $\alpha = 90\%$) As the voting recommendation we take $c^* = arg\ min_{c \in \mathscr{C}}\ votes(c)$, but only for the case that $votes(c^*)/z \geq \alpha$. In case of large belief spaces we relaxed full enumeration to every k-th generated world. With $k = 5$ this lifted the size of the belief space considered for voting from 3 to 15 thousand states.

In *world search* (*Weltensuche*) only one card suggestion is extracted from searching a tree that branches on both a) the location (distribution of free cards into different hands and the skat) and b) the choice of cards to be played.

The time spent for the analyses shows that it may be difficult to search the optimal scoring card on a full hand in a tournament, but it may become feasible to decide whether a game is won against all odds. We applied world search in smaller belief spaces, after some cards have been played already. This *worst-case* variant of *world search* was called *paranoia search* and is especially effective for the declarer, as s/he usually knows the skat, and has a significant information advantage over the two opponents, leading to a much smaller belief space, so that the search can also be invoked earlier in the search. The handling of the more complex knowledge in worst-case opponent worlds search is involved. For both cases, knowledge inferred can be progressed down the tree, and restored on a backtrack. There is a compromise on the complexity of knowledge representation and reasoning and the speed for traversing the search tree.

We apply *move ordering* for traversing the search tree, as breaking ties can still be relevant to both the search tree size and the card recommendation. For example, the player may know that two cards are equally good, but the one hides some information better and may lead to inferior play of the opponents, maybe having less information. At first, we applied heuristic guidelines like the length of a suit, then, we applied different ordering principles based on trump and suit play, as well as the current lead in the current trick. Breaking ties may also be relevant for following playing conventions among the opponent players e.g., to indicate which cards s/he does not have. Finally, we used the suggestion of the expert recommendation module as the first branch in the search tree. As some of these aspects are crucial to the performance of the system, we apply it only to the top levels of the search tree.

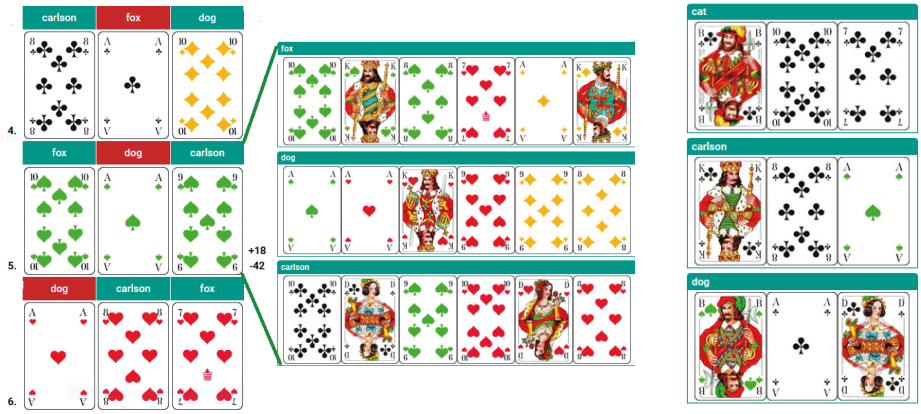


Figure 13.12: Two examples of a successful and one example of an unsuccessful hope card. We see the players' hand cards corresponding to the critical trick. In the left example the context of played tricks and scores is provided with trick 5 being critical.

13.9 Hope Cards

Roughly speaking, a *hope card* may not win most possible worlds, but a large fraction of the winnable ones. The concept applies to many, if not all trick-taking games. Professional players interpret a hope card as a card, which is the only one that leads to victory and exploit the concept, e.g., in issuing a *sharp 10*, which is played in the hope that the corresponding A is owned by the partner, and that the declarer has to obey the suit with a small card, see Figure 13.12 (left, trump is clubs). For another example of a hope card, consider an endgame situation as shown in Figure 13.12 (right). Assume three remaining cards for each player as shown, and that dog as the declarer has collected 40 points. Moreover, let ♠ *A* known to be in the hand of carlson and the location of the other cards being unclear. If cat issues ♣ 7 and carlson continues with ♣ *K*, then there is only one way that dog can win, if he plays ♣ *Q*, and tries to catch ♣ 10.

A *hope card* $c \in \mathscr{C}$ is a recommendation for player $p \in \mathscr{P}$; either a *sole hope card* the only card $c \in \mathscr{C}$ that wins a world in the belief space, i.e., $\exists_{s \in \mathscr{K}_p} legal(p,s,c) \wedge win(p,s,c)$, and $\forall_{s' \in \mathscr{K}_p} \forall_{c' \in \mathscr{C} \setminus \{c\}} legal(p,s',c') \Rightarrow \neg win(p,s',c')$; or an *f% hope card* is a card $c \in \mathscr{C}$ that wins at least fraction f of all winnable worlds:

$$f \geq \frac{|\{s \in \mathscr{K}_p \mid legal(p,s,c) \wedge win(p,s,c)\}|}{|\{s \in \mathscr{K}_p \mid \exists c' \in \mathscr{C} : \ legal(p,s,c') \wedge win(p,s,c')\}|}$$

Instead of naively enumerating the states in the belief space, for hope cards we filter the ones that win. Hope cards are available for both declarer and opponent play in all three table positions (forehand, middle hand, rear hand). There are also hope cards for Schneider and Schwarz. Note the difference to *high-confidence cards* that have the highest vote, who also apply in worlds that do not win.

13.10 Towards an ELO Rating System

The ELO rating system is an established method for calculating the relative skill levels of players in zero-sum games such as Chess. The playing strength of a Chess player, program, or engine, reflects the ability to win against other players, given by a number. ELO is also used in Video Games, American Football, Basketball, Major League Baseball, Table Tennis, Scrabble and Diplomacy, and many other games.

The ELO rating has been devised for games of skill, usually two-player zero-sum deterministic games with full observability. The difference in the ratings between two players serves as a predictor of the outcome of a

single game. Two players with equal ratings, who play against each other are expected to score an equal number of wins. A player whose rating is 100 points greater than their opponent's is expected to score 64%; if the difference is 200 points, then the expected score for the stronger player is 76%.

The players' ELO rating is represented by a number which changes depending on the outcome of rated games played. After every game, the winning player takes points from the losing one. The difference between the ratings of the winner and loser determines the total number of points gained or lost after a game. If the high-rated player wins, then only a few rating points will be deducted from the low-rated player. However, if the lower-rated player scores an upset win, many rating points will be added to his rating. The lower-rated player will also gain a few points from the higher rated player in the event of a draw, so that this rating system is self-correcting. Players whose ratings are too low (or too high) should, in the long run, get better (or worse) than the rating system predicts and, thus, gain (or lose) rating points until the ratings reflect their true playing strength. The ELO ranking system is highly influential for tournament play. In Chess, for example, a world-wide live ranking list decides on tournament invitations, up selecting candidates for challenging the World Champion. Moreover, evaluating the performance level is useful for tracking progress of players and AIs in online game playing, and to compare the level of play with human strength.

For games that include chance, like card games, however, devising an appropriate ELO ranking system is much more challenging. Due to the randomness in the deal, it takes more games to derive statistical meaningful results. Such a novel ranking system has a huge impact, as it can be used for a evaluating tournaments and set-up chance-reduced, skill-emphasizing, much fairer ranking list of the players, to be awarded for their skill of play. It is very much needed for online platforms, as motivating aspects for the players it helps to producing high-score tables.

In many countries, there are monetary and legal consequences, whether a game is based on skill or chance. For the usual setting of including both, extracting the skill factor of a game of chance is wanted. Existing ranking systems like average scoring values are sensitive to the strengths of the opponent players, which cannot be compared easily to a global ranking. Even when working towards this goal, by looking at the ELO ranking, however, we cannot expect an exact derivation of a probability of winning as done in the original ELO system. Instead, we align the ELO ranking to an accepted scoring system.

We implement a refined ELO system that covers most important aspects of such games, compensating for the strength of the opponents and the factor of chance. We use it for evaluating players performance in the experiments and discuss different design choices and parameterizations. The results are compared to AIs self-play. We exemplify the considerations in Skat, but the system can be adapted and applied to other card games and beyond.

Skat is one example for a game of mixed chance and skill There is a wide consensus that while in a single game the variation due to randomness in the deal is large, good players show their strength eventually. Skat is played in club, national and international tournaments, either online or *over-the-table*, where the winning is determined either by number of wins, accumulated scoring or via the (extended) Seeger scoring system. Tournament results and rating lists often show average scoring results. By the continuous growth in playing Skat, a fair, transparent, and objective ranking for the game is a demanding necessity, however, due to the situation of being a three-person game with unknown card distributions, this is much more complicated than in Chess with its defined starting position. So far, this has only been done based on the average points in the accepted scoring system, and, in the event of a tie, based on the larger number of games won. Even if played for hobby and leisure, what is generally missing is a players' reference value, to match a player to an equally strong table. There are early approaches implemented in some Skat forums, where they take all players and then rank only the players who are advanced according to an internal analysis. There is also the need for a high level of confidence, especially if there are significantly fewer data sets behind. If there are only a few games play a good score can be a matter of simply having good cards.

One has tried to create a reference value using an average scoring, e.g., *25 points / game* for a player, and of course adding some bonus at a higher player level, if you still win against strong opponents as the declarer. What has generally been neglected so far is to include a measure for hand strength of a game in the evaluation. Because a game with 100% winning probability almost everyone can win; but a game only with 50% only the

fewest. There are sufficiently tested, simple and complex winning forecasts for games so that one could receive more points for winning a simpler game than for a difficult one. To allow weaker players a better scoring, some online platforms dampen the scoring of good players by modifying the deal, given worse cards to the stronger players. We strongly believe this is the wrong way of dealing with the problem, hard to make fair and transparent, and players experience this as being closer to a fraud. It is better to include the handicap into the ranking function, so that weaker players are awarded if they win against stronger opponents.

We want an ELO score aligned to the Seeger-sytem of DSKV/ISPA, which compensates for the factor of chance, and normalizes with respect to the strength of the opponents. This should lead to a wide acceptance for Skat enthusiasts, tournament directors, and, finally, the Skat players. We require to include the factor of chance as a surplus or deduction wrt. estimated chance of winning; to cope for the value describing the *game size*, i.e., the mean value against a averaged comparison value, computed once for a series; an ELO list adapted accordingly – for a starting value, every series of 36 games 3 new ELO numbers are computed; if the player plays against strong opponents s/he gets rewarded for the risk taken, while if s/he plays against weaker ones, s/he receives less bonus; to choose parameters to reach an asymptotic 1100 extended Seeger Pts/series for a good Skat player, and about 650 Pts/series for a bad player; a quick evaluation of thousands of games to compared player performance in ELO rating list.

We are aiming at a fair evaluation of the playing strength with reduced components of chance.

13.10.1 Scoring Systems and Hand Strength

The extended Seeger (aka Seeger-Fabian) system is an established scoring system for Skat to evaluate a series of X games. Usually, we have $X = 36$. Folded games, for which there are no bids, are neglected.

The evaluation score is based on the number of wins and losses of each player in the series, and the game value oft the games being played. For a single game g of a player P, the outcome is $V(P,g)$, if the game is won for the declarer, and $-2 \cdot V(P,g)$, if it is lost. In a series of games $G = g_1, \ldots, g_k$ these values are added for each player, so that $V(A,G) = V(A,g_1) + \ldots + V(A,g_k)$.

For example, we have players A, B, and C. In the extended Seeger scoring System the evaluation strength ES is defined as $ES(A,G) = V(A,G) + 50 \cdot (\#wins(A,G) - \#losses(A,G)) + 40 \cdot (\#losses(B,G) + \#losses(C,G))$. Similarly, for B and C we have $ES(B,G) = V(B,G) + 50 \cdot (\#wins(B,G) - \#losses(B,G)) + 40 \cdot (\#losses(A,G) + \#losses(C,G))$ and $ES(A,G) = V(A,G) + 50 \cdot (\#wins(C,G) - \#losses(C,G)) + 40 \cdot (\#losses(A,G) + \#losses(C,G))$. The outcome of a series of 36 games might be as follows $V(A,G) = 273$, $\#wins(A,G) = 8$, $\#losses(A,G) = 1$, $V(B,G) = 152$, $\#wins(A,G) = 12$, $\#losses(A,G) = 4$, $V(C,G) = 495$, $\#wins(A,G) = 11$, $\#losses(A,G) = 0$. Then, we have $ES(A,G) = 273 + 350 + 160 = 783$, $ES(B,G) = 152 + 400 + 40 = 592$, and $ES(C,G) = 495 + 550 + 200 = 1295$. In a set of equally strong players, value ≥ 1000 is the result of strong play. However, it could also be the result of very good cards.

Determining luck in the cards resorts to knowing and comparing with the strength of the hand or, even better, to the estimated winning probability. None of the values are exact, but help to predict the outcome of the game.

Walter von Stegen proposed to evaluate the strength of the given hand using the points system an expert Skat player, youtuber and teacher. It only applies to trump games. Roughly speaking, it counts the number of Js with 2 points, and the number of trump and high-cards (As or 10s) with 1 point (high trump cards are counted twice). Additionally, strong Js and low bids are awarded.

Thomas Kinback is a professional Skat player, teacher, and author. He developed another counting system for hand strength. The hand strength value differs in grand and suit games and includes a measure for tricks going home. Moreover, different configurations of cards in a suit as well as being in forehand get a surplus.

We used an accurate prediction of the declarer's probability to win. More precisely, we devise a winning probability $Prob(h,s,t)$ including the Skats s together with hand h. These probabilities are used in the first

stages of the Skat game: bidding, skat taking and game selection, and skat putting. The expert games can easily be analyzed statistically, which we do for all kinds of games being played. There is, however, a fundamental difference in estimating the winning probability in null, and trump games. For null games, we estimate the winning probability $Prob(h,c)$ in each suit c separately. The winning probability, then, is $Prob(h) = Prob(h,clubs) \cdot Prob(h,spades) \cdot Prob(h,hearts) \cdot Prob(h,diamonds)$. For trump games, we consider the so-called winning parameters: Free suits: the amount of non-trump suits that the player does not have in hand; *skat value*: the number of eyes in the skat, condensed into four groups; *bidding value*: the three different final bidding values in the bidding competition, summarized into four values; *declarer's position*: the position of the soloist in the first trick; *trump length*: the number of trump cards in hand; *non-trump aces and tens*: the number of non-trump high cards in hand; *Jack group*: set of jacks condensed into groups; *lost cards*: number of tricks expected to be lost;

Through statistical tests we were able to validate that these parameters have a significant influence on the profit or loss of the expert games and can therefore be used as essential attributes to accurately assess the probability of winning a grand or suit game. More precisely, the grand table is based on 7 to 8 of these winning parameters and the color game table on 9 winning parameters.

13.10.2 ELO System for Zero-Sum Games

Two-Player Games. The ELO ranking system for two players in a zero-sum game works as follows. If Player A has a rating of R_A and Player B has a rating of R_B, the formula for the expected score of Player A is $E_A = 1/(1+10^{(R_B-R_A)/400)})$ Similarly, the expected score for Player B is $E_B = 1/(1+10^{(R_B-R_A)/400)})$. Hence, for each 400 rating points of advantage, the expected score is $10\times$ the opponent's one. When a player's tournament scores exceed their expected scores, the ELO system takes this as evidence that player's rating is too low and needs to be adjusted upward. Similarly, when a player's tournament scores fall short of their expected scores, that player's rating is adjusted downward.

Elo's original suggestion, which is still widely used, was a simple linear adjustment proportional to the amount by which a player over-performed or under-performed their expected score. The maximum possible adjustment per game, called the K-factor, was set at $K = 16$ for masters and $K = 32$ for weaker players. Supposing Player A was expected to score E_A points but scored S_A points. Updating that player's rating is by setting $R_A \leftarrow R_A + K \cdot (S_A - E_A)$.

Multiple Player Games. When two players have ratings R_A and R_B, determined via the ELO model for two players, we have computed their expected scores against each other: $E_A = 1/(1+10^{(R_B-R_A)/400})$ and symmetrically: $E_B = 1/[1+10^{(R_A-R_B)/400}]$ Their ratings are updated by comparing the result of the game with what was expected, e.g., $R_A \leftarrow R_A + K \cdot (S_A - E_A)$, where S_A is the score of the game (1 for a win, 0 for a loss, .5 for a draw, etc.) and K is a *volatility factor* that is equal to the maximum number of rating points that can be gained/lost in a game. Symmetrically, whatever was added to R_A is subtracted from R_B.

One can rephrase ELO the following way: The probability of a player winning the game is modeled as being proportional to $10^{rating/400}$. Of course, you need to normalize the probabilities by dividing by the sum of these numbers. This is the same formula as before: $E_A = 10^{R_A/400}/(10^{R_A/400} + 10^{R_B/400})$. Dividing numerator and denominator by $10^{R_A/400}$ we get $E_A = 1/(1+10^{(R_B-R_A)/400})$. This rephrasing is a lot easier to expand to more players: Each one has a probability proportional to $10^{rating/400}$ of winning the game. $E_A = 10^{R_A/400}/(10^{R_A/400} + 10^{R_B/400} + 10^{R_C/400})$. The update rule does not need to change.

Player (anonymous)	Number of Games	Aver. Score (per Game)	Average Score (per 36 series)
e15352892b7a6cd14e7dedaff97672ec	195	30.75	1107.15
8afc83bd59fa1ed31b88d6d4f12286be	89	30.11	1084.31
18b056ef4c0a2b11464af83bcf69b98d	207	29.90	1076.58
⋮	⋮	⋮	⋮

Table 13.1: Average scoring values.

13.10.3 ELO System for Games of Chance

We came across the topic of devising an ELO scoring system when playing on Michael Buro's Skat platform, which offers a ELO-type of ranking and when we asked him about certain peculiarities. He couldn't remember how that number had been defined exactly, but we got some impressions on its pros and cons. One thing had to be given to his ELO numbers: they express the table strength situation relatively well, depending on whether you were winning or losing against stronger or weaker opponents (in terms of their ELO number). The value of the cards, however, was not included in the computation, which is always a drawback with long streaks of bad luck or luck. Assessing the skill level based on the points scored in relation to the value of the cards turns out to be an important aspect, such a corrected analysis says a lot about the true and luck-adjusted performance of a player. Therefore, we find it fundamentally interesting to consider an ELO number as discussed above as a supplementary, alternative rating number. The methodology how this is calculated in games of no chance like Chess is well understood, but there are some clear differences anyway, so that we introduce a modified algorithm.

Including Opponent Strength. For a better assessment, an ELO rating system is needed, which combines playing results, strength of the opponents and luck in card deals. Unfortunately, a widely accepted rating system does not exist for Skat, nor are other means of measuring playing strength used on selected servers accessible. By the influence of chance computing ELO is more challenging than for Chess and similar deterministic games: there is no doubt, that even much weaker players can beat much stronger ones rather easily when given a very good hand of cards.

Other ELO systems use a freely chosen starting rating, so one can base individual evaluation of a player to consider the ELO development of previous opponents. We think that keeping all previous series individually for each player is not necessary. Due to the calculation, the influence of the "old" results fades continuously anyway and one, therefore, our system does not have to reevaluate the series that were played long ago.

We compute the sum of Seeger-Fabian evaluation of played series $S = S_0 + S_1 + S_2$ and of the rating $R = R_0 + R_1 + R_2$. Next, we calculate the expected values $E_i = R_i \cdot S/R$ for $i = 1,2,3$. This has the effect that the sum of expected value is equal to the series and expected values are in the same relation as the rating of the players. This way outliers are avoided. The update then is $R_i \leftarrow R_i + K \cdot (S_0 - E_0)$ $i = 1,2,3$. As an example consider the rating values $R_0 = 1500$, $R_1 = 750$, $R_2 = 750$ (assuming a ELO rating sum of 3000 and a start value of 1000), and a series with an actual scoring of 1200, 800, 800. Then $E_0 = 1400$, $E_1 = 700$, and $E_2 = 700$, The update yields new ranking values $1500 - K \cdot 200$, $750 + K \cdot 100$, and $750 + K \cdot 100$. If we take $K = 0.02$, then the new rankings are $R_0 = 1496$, $R_1 = 752$, and $R_2 = 752$. We see that it is not necessarily advantageous for the better player to play against weaker ones. In contrast to his/her clear victory he actually loses rating points, as s/he under-performs to the expectation. In our initial experiments shown in Table 13.2 value $K = 0.02$ leads to good results. We see that K has a significant influence not only on the spread of values but also on the final standing of the players.

Small-Sized Series. If due to folding (no one bids), interrupts (people leave the table) or different chosen series size (e.g., due to time limitation), a series may not always consist of the assumed number of games (36 in our case). One option to include the results into the ELO computation is rescaling the individual score values by the ratio α of played games and requested games, so that $S_i = S_i \cdot 1/\alpha$, and also limiting the update effect to $R_i \leftarrow R_i + \alpha \cdot K \cdot (S_0 - E_0)$, $i = 1,2,3$.

S_0	S_1	S_2	E_0	E_1	E_2	R_0	R_1	R_2
1387	1018	152	852.33	852.33	852.33	810.69	803.31	785.99
501	1359	934	943.78	935.19	915.03	801.84	811.79	786.37
637	1284	1125	1017.67	1030.30	998.04	794.22	816.86	788.91
800	812	913	835.59	859.41	830	793.51	815.92	790.57
1213	1221	965	1123.81	1155.54	1119.65	795.30	817.22	787.48
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 13.2: Evolution of ELO values with start value 800.

Pos	#Games	Sum of Values	Mean
1	29930	1264945	42.3
2	26708	1069177	40.0
3	27206	1101191	40.5
1–3	83844	3435313	**41.0**

Table 13.3: Computing average game value.

Reducing Effect of Chance in the ELO System. When talking to Skat experts we identified two main factors that are frequently addressed in assessing the luck of players in card games: strong hands and high-valued games.

Hand Strength. For integrating chance into this ELO system one extreme is to avoid evaluating games with 100% winning probability. Instead, we normalize the strength of the current hand per game category with the mean hand strength (using winning probabilities, the Kinback, or the more involved von-Stegen system). E.g., if the mean is 8.72 and the current hand is 9.5 we have a correction term of $c = 1.09$, so that a score of 86 points is downgraded to 79. If the values c fall outside the interval $[0.5, 2]$ we have clipped it to avoid extreme behaviors by setting c = $\max\{\min\{c, 2\}, 0.5\}$. Now every game is re-weighted. Assuming 875 ELO for the declarer and 950 for the opponents, we derive another normalization factor of 1.086. Taken together, so that a score of 86 will be downgraded to 73.

High-value Games. There is another objective often attributed. In some series people score high with playing only a few, but very high-valued games (usually Grands), for the others they do not bit. To avoid such over-exaggeration, one could adapt the score of the games. To bypass this influence of chance we calculated the mean point score $\bar{x}$ of a game with 41.0 (see Table 13.3).

Instead of the actual (or normalized) game value influencing the ELO score, for this chance breaking option we take the (normalized) game value $\bar{x}$ (or normalized, $\bar{x}/q$). This might be seen as an extreme setting, as giving lower- and higher-valued games the same influence in the ELO formula, does not reflect the risk taken to achieve the higher score. Of course, many combinations of two adaptations to reduce the element of chance in the game are possible. In both cases, we use the adapted score $s'_i = c_i \cdot s_i$, for each player $i = 0, 1, 2$, and sum the values for a series score values S'_i. The S_i's are then into the above ELO update formula. We have $R_i \leftarrow R_i + K \cdot (S'_i - E_i)$, for $i = 1, 2, 3$.

Including Start ELO. Finding an appropriate initial ELO ranking value. Voices differ between lower value of 800 and a higher value of 1000 as a valid choice. An alternative is to use the average Seeger score of some n first played tables. In other words, for player i use the initialization $R_i^{(n)} = \sum_{t=1}^{n} S^{(t)}/n$ (exponent shows the index of the table). To initialize the ELO ranking further games could also be considered, possibly after some rescaling to the Seeger score.

Newcomer Progression. For the ELO numbers it is well-established fact that new players get a larger K, value to be able to grow in their experience and climb up the ranking more quickly. With a larger number of games and ELO value, the factor then decreases (from say $K = 40$ to $K = 10$). This also compensates for a wrong assumption on the playing strength in the initial ELO.

13.11 Experiments

The Skat AI is written in C++. Each player client runs on 1 core.

Studies. For Grand (after taking skat) ♠J, ♡J, ♡10, ♡Q, ♡9, ♡8, ♣A, ♣10, ♣9, ♣7, ♡7, ♢J world search finds the optimal declarer score of 61 in 5m5s (1m24s in the decision variant) by putting ♣7 and ♣9, and first issuing ♣A. If not cut, ♣10, followed by ♡7. One worst case play for the declarer is ♡8, ♡A, ♠A; ♡9, ♡A, ♠A; ♡10, ♣J, ♠10 with 59 eyes. The decision variant took 1m23s, also returning ♣A.

For hand ♣J, ♠J, ♡J, ♢J, ♣9, ♣8, ♣7, ♠9, ♠8, ♠7 with skat ♡7, ♢7 a game in clubs is won for the declarer in forehand position, but potentially lost in middle and in rear hand positions. Solving all three decision variants takes 16s in total and returns ♢J, ♠A, and ♠K, respectively. The best scores for the declarer are 63 (fore hand), 48 (middle hand), and 49 (rear hand). The ♢J was found less than 1s, the ♠A in 1m31s, and the ♠K in 3m15s.

Database Replay. During replay we determine the average of the game value according to the extended Seeger-(Fabian-)System, the internationally agreed DSKV standard for evaluating game play, normalized to a series of 36 games. The score is based on the number of wins and losses of each player in the series, and the game value of the games being played. For a single game g, the outcome is $V(D,g)$, if the game is won for the declarer D and $-2 \cdot V(D,g)$, if it is lost. In a series of games $G = g_1, \ldots, g_X$ these values are added for each player, so that $V(A,G) = V(A,g_1) + \ldots + V(A,g_X)$. The evaluation strength of Player A wrt. B and C is $V(A,G) + 50 \cdot (\#win(A,G) - \#loss(A,G)) + 40 \cdot (\#loss(B,G) + \#loss(C,G))$.

The obtained results on 50,000 human expert trump games are presented in Tables 13.4– Table 13.6. For the three valid combinations of AI/Human bidding/discarding/game announcement we separate between the play with and without the support of Paranoia search. In the columns we further partition the game outcomes with respect to the declarer in the *i*) original Human game play, *ii*) an open card solver (that we call Glassbox), and *iii*) AI trick-taking selfplay.

In Table 13.4 we see that with the support of the Paranoia search, the declarer can win $42,228 - 41,765 = 463$ more than the AIs without Paranoia search and far more than the Humans in their play $42,228 - 41,283 = 945$. This is a significant progress, given that the number of wins was already high. The number of games won (and the extended Seeger values) were higher than the ones obtained by the humans.

The AIs with KBPS show better winning ratios than the humans, and a significant positive effect on the playing strength in extended Seeger score: for AI bidding and Skat putting almost 1,000 points. With up to 50% additional time, there is a computational trade-off, but in server play selecting the card to play remains below 5s. In contrast to Null games, automated Skat putting in trump is worse to the Human one, and, therefore, subject to further research. For AI bidding the total of wins/losses is not matching the total number of games, as some games might be folded.

We analyzed another set of over 75 thousand human expert games (all kinds of games). We varied the card number k to start approximate KBPS at card k and KBPS at card $k+3$. At $k = 6$ we reached 1000.24 extended Seeger scoring points. At $k = 3$ we could slightly improve the value to 1001.39, in a tradeoff of a slowdown factor 2-3.

ELO Ranking of Human Players. The ELO ranking routine is embedded in our Skat AI. As the formulas are straight-forward, we believe that a Skat web application with a world-wide ranking list (like `2700chess.com`) can be derived.

As input, we selected a database of 83844 expert games, played on a server in 2329 series (tables) of 36 games each. The data we process has anonymous (alias hashed, blind) player names and table indices, to avoid options for giving advantage to known Skat players.

For this series of human games, we derived a proper ELO ranking, according to the above formulas and different parameterization for K, the hand strength function, or the inclusion of reducing the effect of card luck. On our computer this analysis takes about 4s mainly due to computing the winning probabilities. The outcome is an ELO high-score table. Tables 13.7 and 13.8 show the derived list.

	Human Wins	Glassbox Wins	AI Wins	+Paranoia Opponents	−Paranoia Opponents
−Paranoia Declarer	false	false	false	2,563	2,530
	false	false	true	2,208	2,241
	false	true	false	231	226
	false	true	true	2,658	2,663
	true	false	false	3,438	3,407
	true	false	true	5,540	5,571
	true	true	false	975	970
	true	true	true	31,285	31,290
Total +PO	41,283	35,149	41,691	48,898	-
Total -PO	41,283	35,149	41,765	-	48,898
Total Score				977.76	980.07
Total Time				37h:51m	31h:01m
+Paranoia Declarer	false	false	false	2,460	2,458
	false	false	true	2,315	2,313
	false	true	false	193	194
	false	true	true	2,696	2,695
	true	false	false	3,236	3,240
	true	false	true	5,742	5,738
	true	true	false	785	788
	true	true	true	31,475	31,472
Total +PO	41,283	35,149	42,228	48,898	-
Total -PO	41,283	35,149	42,218	-	48,898
Total Score				990.25	991.79
Total Time				43h:32m	42h:06m

Table 13.4: Skat AI Replaying 50,000 Human Trump Games with and without KBPS, using AI Bidding Game Selection and Skat Putting. Score is extended Seeger, averaged over 36 games. Table split based on KBPS being applied for declarer and opponents. Total of games is smaller because of $1,102$ foldings (no bid).

	Human Wins	Glassbox Wins	AI Wins	+Paranoia Opponents	−Paranoia Opponents
−Paranoia Declarer	false	false	false	3,854	3,822
	false	false	true	2,779	2,811
	false	true	false	198	195
	false	true	true	1,040	1,043
	true	false	false	2,137	2,110
	true	false	true	5,271	5,298
	true	true	false	965	953
	true	true	true	33,756	33,768
Total+PO	42,129	35,959	42,846	50,000	-
Total−PO	42,129	35,959	42,920	-	50,000
Total Score				953.31	955.91
Total Time				34h:20m	25h:31m
+Paranoia Declarer	false	false	false	3,784	3755
	false	false	true	2,849	2,878
	false	true	false	184	128
	false	true	true	1,054	1,056
	true	false	false	1,998	1,973
	true	false	true	5,410	5,435
	true	true	false	761	752
	true	true	true	33,960	33,969
Total + PO	42,129	35,959	43,273	50,000	-
Total - PO	42,129	35,959	43,338	-	50,000
Total Score				963.35	965.53
Total Time				47h:21m	38h:05m

Table 13.5: Skat AI Replaying 50,000 Human Trump Games with and without KBPS using Human Bidding, Game Selection, and Skat Putting. Score is extended Seeger, normalized to 36 games. Table split KBPS being applied for declarer / opponents.

ELO Ranking of Skat AIs. We have developed an AI for the Skat game, capable to beat even advanced club players. So far, we have recorded a few mixed Human-AI games. We could replay all the games with our Skat AI. To accelerate the evaluation on multiple cores, we partitioned the input file into 9×8388 games and 1×8352, and started the driver for the AIs 10 times. The AI wins $4132 + 4768$ games that the Humans do not, while the Humans win $5273 + 1085$ that the AIs do not. This suggests that the AIs are showing a better playing performance.

	Human Wins	Glassbox Wins	AI Wins	+Paranoia Opponents (PO)	-Paraonoia Opponents (PO)
−Paranoia Declarer	false	false	false	3,941	3,894
	false	false	true	2,519	2,566
	false	true	false	200	196
	false	true	true	1,211	1,215
	true	false	false	2,611	2,581
	true	false	true	5,502	5,532
	true	true	false	934	923
	true	true	true	33,082	33,093
Total +PO	42,129	35,959	42,314	50,000	-
Total -PO	42,129	35,959	42,406	-	50,000
Total Score				935.23	938.41
Total Time				34h:10m	25h:15m
+Paranoia Declarer	false	false	false	3,853	3,808
	false	false	true	2,607	2,652
	false	true	false	187	185
	false	true	true	1,224	1,227
	true	false	false	2,452	2,415
	true	false	true	5,661	5,698
	true	true	false	732	724
	true	true	true	33,284	33,292
Total +PO	42,129	35,959	42,776	50,000	-
Total -PO	42,129	35,959	42,869	-	50,000
Total Score				947.41	950.39
Total Time				37h:39m	46h:49m

Table 13.6: Skat AI Replaying 50,000 Human trump games with and without KBPS using Human Bidding, and AI Skat Putting. Score is extended Seeger normalized to 36 games. Table split on KBPS being applied for declarer / opponents.

No	P-No	Player Name (hashed)	Elo
1	P-2	550d2996ded442172116845433d09570	856.77
2	P-25	bb01edd8abe48241a43b81af3895c925	854.80
3	P-12	542e18999d44ec8427481bf9b75c4bc6	853.42
4	P-27	693730a9db9143b7fc0a37d948b6fb76	853.32
5	P-31	e243ef566bac2607033c320e56ca3c	853.06
⋮	⋮	⋮	

Table 13.7: ELO ranking for Skat on sample with $K = 0.02$.

No	P-No	Player Name (hashed)	Elo
1	P-368	eea6ea6035503ac4493cf46f12ced5b0	1063.61
2	P-616	7c36b944da8dcab82ed1e8b1822438cf	1058.20
3	P-663	6d71ec67d159864149195c79f2cd289f	1055.64
4	P-60	fcfcbaa041c41d65ea2f0c64070f3af5	1017.28
5	P-26	aa6ee7912aca31ea9d26371b539f3b43	1002.74
⋮	⋮	⋮	

Table 13.8: ELO ranking for Skat on sample with $K = 0.05$.

AI self-play of same bots does hardly help to illustrate an ELO evolution. If there are three bots, which are all the same, the increase and decrease is essentially the same, given that the sum of the ELO values stays constant. To show progress, AIs of different strengths have to play each other or against humans. The ELO ranking for logged series of human vs AI server games automatically derives high-score rankings. It indicates that our AI is playing well against good club players. This implementation can evaluate Human, mixed, or AI tournaments.

Sensitivity Analysis. To study the change in the ELO distribution we varied the parameters for the elements of change i_1 (winning probability) i_2 and the K-factor. Table 13.9 shows that the distribution of ELO values for a growing K widens, so that the volatility factor K can be adapted to the needs. For a world-wide ELO list, or for frequently played online platforms, a smaller value of K might be appropriate for a moderate increase and decrease in value. For a tournament or club championship, a larger value of K will be better. We see the effect for reducing the element of chance in the ELO numbers. We are still trying to measure the effect statistically.

Visualization of Individual ELOs. Figure 13.13 illustrates the dynamic change in the ELO scoring values for some selected players. For the time-contracted view, only the changes of ELO-values are stored, for the

i_1	i_2	K	Max	Mean	Min	10	25	75	90
0	0	.01	924.84	849.91	790.21	862.18	854.33	845.07	839.22
0	0	.02	976.54	849.90	739.95	874.07	858.67	840.34	829.28
0	0	.04	1038.27	850.33	652.66	893.69	866.58	831.44	810.01
0	0	.08	1149.71	850.81	532.15	922.17	881.78	815.78	778.33
0	1	.01	920.87	849.87	793.42	860.88	854.25	845.61	840.24
0	1	.02	962.12	849.75	745.01	870.80	858.32	841.32	831.08
0	1	.04	1018.33	849.49	664.03	889.39	866.21	833.93	814.13
0	1	.08	1120.17	849.02	542.22	920.97	881.58	819.69	785.47
1	0	.01	927.56	849.85	776.35	863.45	854.95	844.96	838.80
1	0	.02	981.48	849.91	714.23	875.16	859.43	840.14	828.00
1	0	.04	1045.39	850.27	617.78	893.19	867.97	830.71	808.44
1	0	.08	1165.40	850.29	502.82	929.80	884.76	813.70	773.46
1	1	.01	922.10	849.81	777.90	862.79	854.83	845.28	839.55
1	1	.02	973.63	849.67	715.30	874.93	859.47	840.70	829.59
1	1	.04	1036.14	849.30	614.48	895.34	868.56	832.52	809.81
1	1	.08	1144.11	849.79	487.55	931.75	884.90	817.03	778.09

Table 13.9: ELO distribution for chance and change parameters.

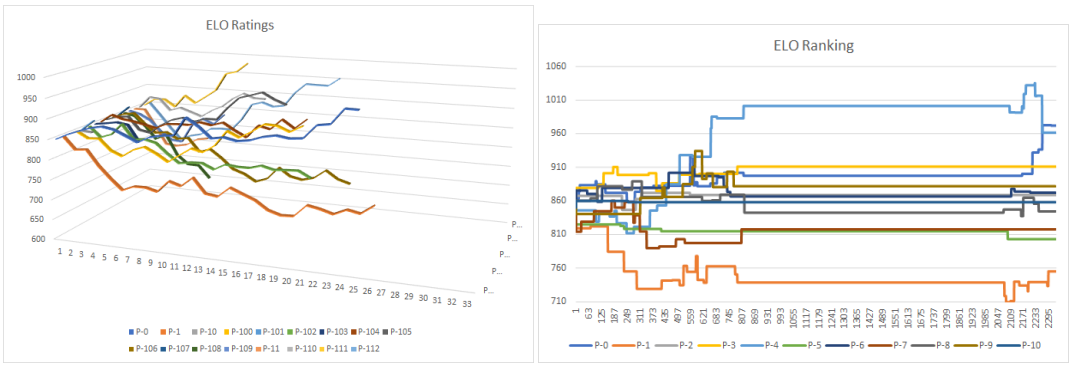


Figure 13.13: Players' ELO rating: time-contracted view (left), time-expanded view (right).

time-expanded view, the table ID of the game is taken to show the evolution of ELO of different players over time.

This visualization option could be provided for the players (or tournament directors). Based on a mixed tournament and ELO rating, one could also award titles like National-, International-, or Grandmasters, as done in Chess. In a time-contracted view, Figure 13.14 shows the evolution of the change in the ELO values. In the longer series we see that when including the ELO formula adaptation for chance, the playing strength values show smaller amplitudes, i.e., the curves become a bit smoother. This means that the real strength of becomes visible faster than without the inclusion of the chance parameters.

Win-Rate and Accuracy. Judging self-play of three identical players is demanding, as applying the ELO rating system is not applicable, given that the average ELO of the play will not change.

Let $won(human, opencard, ai)$ be the numbers of games won in an experiment, with $human, opencard, ai \in \{0,1\}$, and $total = \sum_{(i,j,k)\in\{0,1\}^3} won(i,j,k)$. We use the following criteria to measure playing strength (*the higher the better*).

Win Rate the number of games won by the declarer divided by the number of played games (all but folded ones), i.e., Win-Rate $= (won(0,0,1) + won(0,1,1) + won(1,0,1) + won(1,1,1))/total$.

Accuracy Fraction of games matching the prediction of the open card solver (OCS), i.e., Acc $= (won(0,0,0) + won(0,1,1) + won(1,0,0) + won(1,1,1))/total$.

Combination Win Rate and Accuracy Optimizing the win rate leads to improvements of the declarer but not the opponents, which are better reflected with OCS accuracy. The closer the play is to the open-card solver, the better the opponent players communicate on their respective knowledge of the cards. To combine the two we use Combined $= (\text{Win-Rate} + 1 \cdot \text{Acc})/2$,

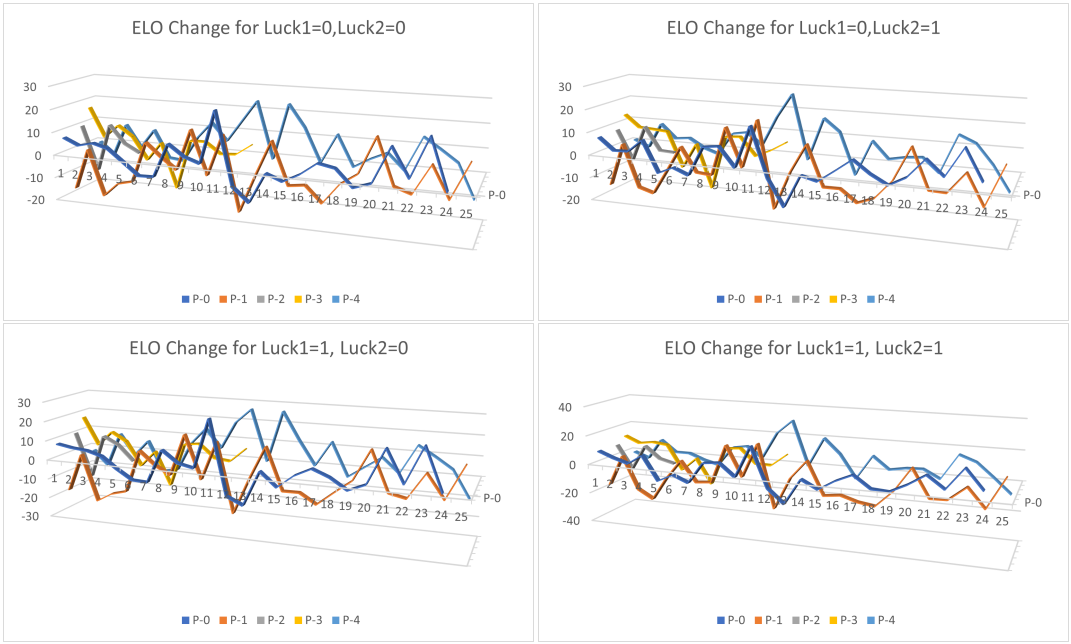


Figure 13.14: Sensitivity analysis varying parameters for reducing effect of chance.

Game	WonAI	LostAI	%Win	# Games
9	6,601	1,600	80.5%	10.04%
10	9,125	2,123	81.1%	13.77%
11	11,120	2,550	81.3%	16.74%
12	15,146	3,573	80.9%	22.92%
23	2,175	1,038	67.7%	3.93%
24	22,127	1,029	95.6%	28.35%
35	39	3	92.9%	0.05%
46	2,865	317	90.0%	3.90%
59	230	3	98.7%	0.29%
	69,428	12,236	**85.0%**	Winrate
Seeger-Fabian: **951.18**			Acc: **84.5%**	

Table 13.10: Final AI performance; Combined = **84.75%**.

Table 13.10 show our final results with further changes to bidding and trump play.

Human-Machine Comparison. In online play is crucial to have a transparent algorithm to generate a fair deal. We use the Mersenne-Twister from the std library to shuffle the cards and validated that 10 million random deal match the theoretical predictions implied by the hyper-geometrical distribution [300]. Their generation is simple enough to show the entire source code.

```
std::mt19937 gen;
for (int k=0; k < 10000000; k++) {
  unsigned int card[32], c = 0, bit = 0;
  for (int i=0;i<32;i++) {
    do { card[i] = gen() % 32; bit = 1<<card[i];
    } while ((bit | c) == c);
    c = c | bit; }}
```

To our surprise, the difference to the mathematically calculated card distribution was smaller than by using overhand shuffling or real random numbers.

Our AI played a online match of 20 series (a total of $19 \times 36 + 1 \times 6 = 690$ Games) in Feb/Mar 2022 against Rainer Gößl, a top Skat player from Germany. In international tournaments at least 10 series of 36 games are

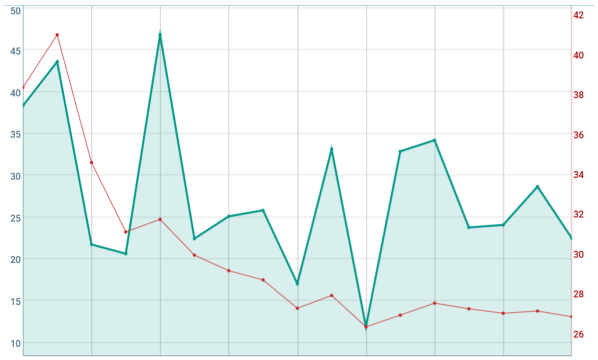


Figure 13.15: Server statistics of top human Skat player playing a tournament against two (identical) AIs: 20 series were played over the period of one month: x axis is time, y-axis the Seeger-Fabian score (per series, green, and cumulative, red). The average Seeger score of was 24.2.

played. In the statistics displayed in Fig. 13.15 we see the human player's performance goes down, suggesting an increase of the program's playing strength during this time span.

About 28.4% of the games were taken on as declarer, 170 were won, and 26 lost. On average one game was folded per series, the average playing time per series was 21.2 min. In this series the AIs performed much better than Rainer Gößl with his score $24.82 \cdot 36 = 893.52$ in the Seeger-Fabian system, and the first engine where he scored less than 1,100 points. Even though the performance is remarkable, it has to be dealt with care. In 20 series there is still a large factor of luck in the cards. In earlier matches, Rainer Gößl achieved better scores against our AIs and against human players. The match was played over a month and not controlled by an arbiter.

13.12 General Imperfect Information MiniMax Search

We now turn to general trick-taking card games. We have experimented with Bridge and Belote, initial implementations of Doppelkopf, Tarot, Spades, Hearts, and many others. The general idea (after some bidding stage) is to have a vector of players, arranged into teams that count their trick scores. Algorithm 13.2 captures the structure of a MiniMax tree search algorithm for such games, invoked from the *chooseCard* driver with the current game state s, current hand h, vector set *parent*, score vector α and set of worlds W. The *team* is linked to the player's *id*. As algorithmic primitives we have

- $<_i$ is a partial order, representing the preferences of player i for pruning;
- $init_i(s,W)$ initializes the result for a player i and must be a lower bound to the actual value (depending on the worlds in W) of the node s for $<_i$;
- $\max_i$ is computing the return value based on the values of the children. It is a fusion of the values of the children rather than a maximum over $<_i$, and should return the correct value of a node s constructed with the values of its children;
- *generateWorlds*() generates the worlds over which the values will be computed according to the knowledge of the players;
- $criterion_i$ a total order over $\mathcal{V}$ representing the preferences of player i for the final choice of the card.

Now that we have a generic algorithm to solve the problem with reasonable time complexity, we will compare different realizations of this algorithm. Each one is applicable for two teams. The score of a final state is the score made by the Max (declarer) team. The first one is a direct extension of usual $\alpha\beta$ pruning algorithm for

Algorithm 13.2: General MiniMax search tree procedure to choose a move for a player in game state s with hand h.

```
tree(s, α, W, parent)
parent[team] ← id
r ← init_id(s, W)
for all c ∈ ⋃_{w∈W} legal(s, w_id) do
    α[team] ← max_id(r, α[team])
    if ∃t ≠ team with α[t] <_{parent[t]} r then
        return ⊥_t
    W_c ← {w ∈ W | c ∈ legal(s, w_id)}
    v ← tree((s, c), α, W_c, parent)
    if v = ⊥_t then
        if t ≠ team then
            return ⊥_t
    else
        r ← max_id(r, v)
return r

chooseCard(s, h)
W ← generateWorlds()
for all every team t do
    parent[t] ← ∅
    α[t] ← ⊥_t
return arg criterion_{c∈legal(s,h)}(tree((s, c), α, W, parent))
```

perfect information games. The second one is $\alpha\mu$, and the last one is a mixture of the two. For each case, we will present the instantiation of the functions above. We will not detail the function $init_i$ and use a trivial lower bound over the value of a node, based on the score already made by a team in the trick-taking card game.

13.12.1 Perfect-Information Monte Carlo

The idea of the *perfect-information Monte-Carlo* (PIMC) algorithm is to sample random worlds following the current distribution, which aligns with the aready inferred knowledge; and, then, in each sample to compute a score *as if it was in perfect information* with the alpha-beta search algorithm. For such set of worlds W, the value of a node is a vector of the scores of each individual world in this setting of perfect information for everyone. As we allow general scoring functions, the node value vector, therefore, is an element in $\mathscr{V} = \mathbb{R}^N$.

PIMC fits a game, where after one card is played, everyone shows its hand and plays open. Thus, the generation of the worlds tries capture the initial unknown distribution, and, then, in each one, plays with perfect information. In terms of pruning, for the Max team a value a is greater than b if the score of every world $w \in W$ is greater in a than in b, i.e., for all $k \in \{1,\ldots,N\}$ we have $a_k \geq b_k$. It is symmetric for a Min node, so that in this case for all $k \in \{1,\ldots,N\}$ we have $a_k \leq b_k$.

In this framework, the template function $\max_i(a,b)$ returns the maximum (alias minimum), according to the team of i, of every score of W. With perfect information in each world, the players can choose the best card. To make the final choice for the card to play, as a form of voting, we choose the criterion to prefer the best mean of the score, according to the team. The details are presented in Algorithm 13.3 (left).

Algorithm 13.3: PIMC (left) and $\alpha\mu$ (right)

```
generateWorlds
W ← ∅
for all k ∈ [N] do
    select w from distribution
    W ← W ∪ {w}
return W

max_i(a, b)
r ∈ 𝒱
for all k ∈ [N] do
    if Max team then
        r_k ← max(a_i, b_i)
    else
        r_k ← min(a_i, b_i)
return r

criterion_i(a, b)
sum_a ← Σ_{k∈{1,...,N}} a_k
sum_b ← Σ_{k∈{1,...,N}} b_k
if Max team then
    return sum_a > sum_b
else
    return sum_a < sum_b
```

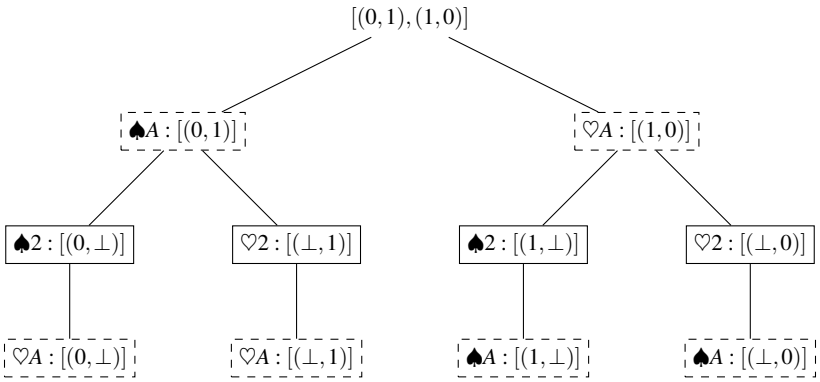
```
generateWorlds
W ← ∅
for all k ∈ [N] do
    select w from distribution
    W ← W ∪ {w}
return W

max_i(X, Y)
R ← ∅
if root then
    R ← X ∪ Y
else
    for all (x, y) ∈ X × Y do
        r ∈ ℝ^N
        for all k ∈ {1, ..., N} do
            if Max team then
                r_k ← max(x_i, y_i)
            else
                r_k ← min(a_i, b_i)
        R ← R ∪ {r}
for all r_1 ≠ r_2 ∈ R do
    if ∀k ∈ [N] r_1k ≤ r_2k then
        if root is Max then
            remove r_1 from R
        else
            remove r_2 from R
return R

criterion_i(X, Y)
if Max team then
    sum_X ← max_{x∈X} Σ_{k∈{1,...,N}} x_k
    sum_Y ← max_{y∈Y} Σ_{k∈{1,...,N}} y_k
    return sum_X > sum_Y
else
    sum_X ← max_{x∈X} Σ_{k∈{1,...,N}} x_k
    sum_Y ← max_{y∈Y} Σ_{k∈{1,...,N}} y_k
    return sum_X < sum_Y
```

Figure 13.16: In this game we follow the rules of bridge except we only play with two enemy players. Trump is clubs. (left) North plays ♣A and South is to play. South does not know if the second card of North is ♠2 or ♡2, it is a 50% − 50% situation. (right) Trump is clubs. North does not know if the second card of South is ♠2 or ♡2. South has to play.

13.12.2 AlphaMu

The problem with PIMC is the hypothesis of perfect information for everyone. It causes several imperfections and difficulties in the choices. An example of a situation, where PIMC fails is Figure 13.16 (left).

Figure 13.17: Application of permutation algorithm to the problem raised in Fig 13.16 (left). The scores are the ones of South, viewed as a Max node. Max nodes are circled by continuous line, and Min nodes by dashed lines. Here world 1 is the world where North has ♠2 (the real world), and world 2 is the one where she has ♡2 (the fake world but South doesn't know it)



In this situation, the expected score of South over these two tricks is 0.5. In fact, South has a 50% probability of throwing the useless ace ♣*A* and to keep the useful ace. In this case South scores 1, otherwise 0. However PIMC would generate several worlds, some with North having the ♠2, other with North having the ♡2, and in each one South would score 1. Indeed if South knows the remaining card of North (this is the case in PIMC algorithm, where perfect information is given), it is easy to keep the right ace, and to win the last trick. Thus PIMC would average the score over the worlds sample and predict a score of 1 for South, that is a mistake.

Algorithm $\alpha\mu$ algorithm is to includes the imperfect information of the player to chose a card into the reasoning. The player at the root of the tree does not know, which world is the good, but the other knows it and plays with perfect information. The algorithm fits a game like Bridge, where one player has to play a card, and everyone else sees all hands. Although it is not perfect in term of information given, it is an improvement over PIMC. We find that with $\alpha\mu$ we achieve a reduction of the information surplus by a factor $(1-\frac{1}{p})$. This is not a surprise, as we removed the information of one player.

Let us illustrate the functions used in $\alpha\mu$. This algorithm solves the issue raised in Figure 13.16 (left). In Figure 13.17 we see that the uncertainty over the outcome of the game is kept. The value computed, $[(0,1),(1,0)]$ represents the fact that one strategy leads to a non-zero score in one world, and another leads to a non-zero score in the other world. In $\alpha\mu$ the call to *generateWorlds* is the same as in PIMC; function $\max_{root}$ computes the union of the values of the children, and removes the elements that are dominated (i.e., if the scores associated to one strategy are all better than an other one, it removes the latter); max_i is the union of the product of strategies of the children. One element is computed by taking one element from all the children (we construct one possible strategy for root), and by taking in each world the best score for i, as in PIMC (if root follows the strategy constructed, then the score would be the one computed, with perfect information of i). Again, the set is simplified by removing the dominated elements for *root*; *criterion* selects the node that contains the strategy with the best expected score over the different worlds in W.

13.12.3 Permutation

Even with $\alpha\mu$ we give away too much information to the players different from root. This can cause mistakes, as shown in Fig. 13.16 (right). If South plays ♠2, she will score 1. If she plays ♣*A* she has 50% probability of scoring 2 and 50% probability of scoring 1. Thus ♣*A* is a better choice. Algorithm $\alpha\mu$ suppose perfect information of North. Thus if South plays ♣*A* she will score 1, and if she plays ♠2 she will score 1 also. Both cards are equivalent, and $\alpha\mu$ could make the mistake of choosing ♠2.

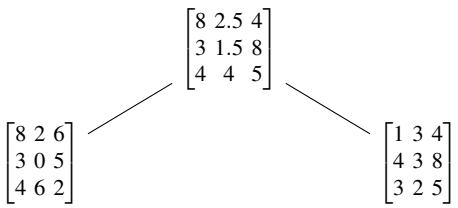
To fix this issue, we need to include imperfect information of all the players in one algorithm. This is the goal of the *permutation* algorithm.

We select the value of a node to be the expected score of each world if everyone played optimally according to its knowledge. At a node with two children, player i will choose the value of the child that has the best expectation, according to its team, over the worlds that are possible according to its knowledge. If the two children have the same expected score, player i has 50% chance of playing each one. Therfore, the value of the node should be a mix of both (the middle of the segment of the scores). If a Max node has two children $[2,5,3]$ and $[3,7,0]$ for instance, its value would be $[2.5,6,1.5]$

The issue with this simple approach is that different players have different knowledge, and different possible worlds. So each set of possible worlds P is different for each player. Moreover, for a player the set of possible worlds depends on its hand, which is unknown. We cannot take one set W for all the tree and suppose it represents all the players, as we had before.

One solution is to compute the values over subsets of W that correspond to the different P possible for a player. For instance, if there are two players Max and Min, each of them can have two hands. Four worlds are possible $\{Max_1Min_1, Max_1Min_2, Max_2Min_1, Max_2Min_2\}$. Suppose Max is the root, and she wants to merge two values at a Min node: $[0,2,3,1]$ and $[2,0,2,3]$. If Min has hand 1, the worlds she will consider are the first and third (she does not know the hand of Max. Thus, the first value is better because it has an expected score of 1.5 instead of 2. But if Min has hand 2 the two values have the same expected score over worlds 2 and 4 that is 1.5. Hence, Min have 50% chance of playing each value and we mix the scores of these worlds. We come up with the value $[0,1,3,2]$. A more detailed example is given in Figure 13.18.

Figure 13.18: Fusion of two values by a Max node. The row correspond to the different hands possible for Min, and the columns the one for Max (m_{ij} represents the score of the world where Min has hand i and Max hand j)



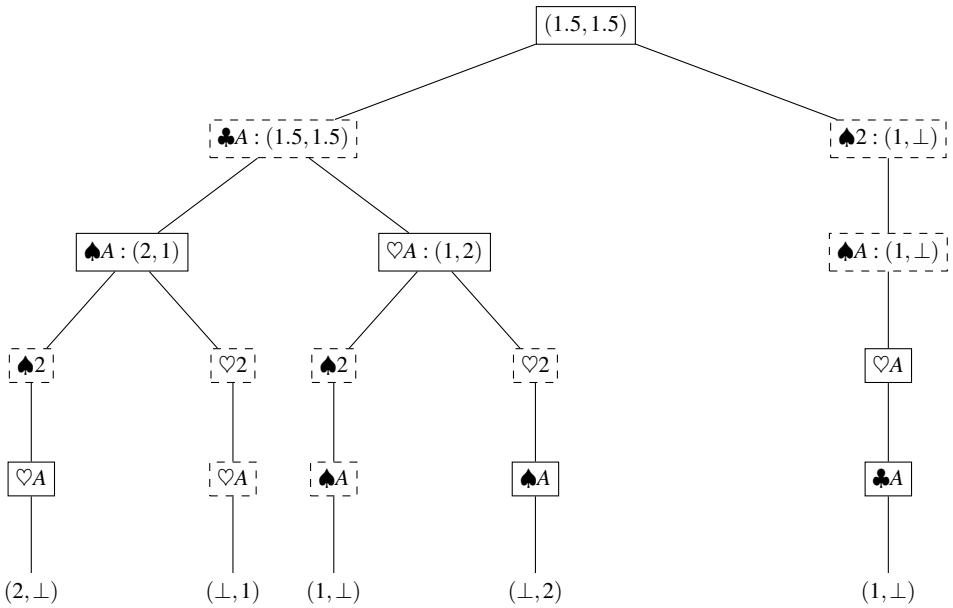
This algorithm solves the issue raised in 13.16 (right) as shown in Fig. 13.19. At the root, South has the choice of two values, over the actual world 1.5 and 1. Because $1.5 > 1$ South will then play ♣A.

To obtain the correct number of worlds for each hand of each player, we require a similarity between the worlds of W. We use a small set of initial worlds and create the set W by taking all possible permutations of the hands with this initial set. Thus, Algorithm 13.4 *generateWorlds* samples worlds and, then, return all the permutations of these worlds; max_i is following the scheme explained before; $criterion_i$ selects the value with the best average score over the possible worlds of i. We prune if all the scores of a value are worse than another one.

For an initial set of experiments we selected two different card games: Belote and Bridge. We implemented the three above algorithms and a random player. We then compared all pairs of algorithms. We assumed that there are two teams, so we attributed one strategy to one, and the other to the second team. We made the matches on 100 different dealings of cards (they are the same over the different experiments). With each deal, we played one time as being dealt, and then exchange the positions of the teams to be fair. We measure the score earned by the team of the first player to play. In each experiment we finally collected the 200 scores over these games. We add up the 100 scores made by one strategy, and the 100 scores made by the other, and deduce the percentage of the total amount of points per game that a strategy wins over the other.

In each algorithm there are some hyperparameters to fix: *n_sample* is the number of world used in the algorithm; *depth_leaf* in $\alpha\mu$ and *permutation* is the depth at which we stopped the specification to end the tree with PIMC;

Figure 13.19: Application of permutation algorithm to the problem raised in 13.16 (right). The scores are the ones of South, viewed as a Max node. Here world 1 is the world where South has ♠2 (the real world), and world 2 is the one where she has ♡2 (the fake world but North doesn't know it)



Algorithm 13.4: Permutation

$max_i(a, b)$

$r \in \mathscr{V}$
for all h possible hand of i **do**
  $score_a \leftarrow \sum_{w \in P_h} a_w$
  $score_b \leftarrow \sum_{w \in P_h} b_w$
  if $score_a < score_b$ **then**
    for all $w \in P_h$ **do**
      if Max team **then**
        $r_w \leftarrow b_w$
      else
        $r_w \leftarrow a_w$
  else
    for all $w \in P_h$ **do**
      if Max team **then**
        $r_k \leftarrow a_w$
      else
        $r_k \leftarrow b_w$
return r

depth_rd is the depth where PIMC stopped to end the tree with the an average of random playouts. For Belote we estimated a 10% approximation of the score, valid with 95% probability with these random playouts. For Bridge, *depth_rd* was not not consistent.

Figures 13.20 and 13.21 show that all algorithms are better than random. We can also observer that $\alpha\mu$ algorithm is slightly better than the others. All these algorithms are designed to play against smart players, thus, the differences of the scores against random are not significant (i.e., PIMC being better than $\alpha\mu$ in beating random in Bridge is not relevant). Permutation is to be slightly worse than the others.

	rd	pimc	$\alpha\mu$	perm
rd	-1.8	-19.3	-22.1	-18.5
pimc	19.3	-3.3	-0.2	1.5
$\alpha\mu$	22.1	0.2	2.4	0.8
perm	18.5	-1.5	-0.8	3.5

Figure 13.20: Scores of team 0 against team 1 in belote. team 0 is the row title, and team 1 is the column title. Here are the hyper-parameters used for every algorithm. Pimc: $n_{sample} = 10$. $\alpha\mu$: $n_{sample} = 10$, $depth\_leaf = 5$, $depth\_rd = 20$. Permutation: $n_{sample} = 24$, $depth\_leaf = 5$, $depth\_rd = 20$

	rd	pimc	$\alpha\mu$	perm
rd	1.5	-25.2	-21.8	-21.6
pimc	25.2	-0.7	-1.8	5.0
$\alpha\mu$	21.8	1.8	-2.1	2.7
perm	21.6	-5.0	-2.7	0.5

Figure 13.21: Scores of team 0 against team 1 in bridge. team 0 is the row title, and team 1 is the column title. Here are the hyperparameters used for every algorithm. PIMC: $n_{sample} = 10$. $\alpha\mu$: $n_{sample} = 10$, $depth\_leaf = 5$, $depth\_rd = 9$.

13.13 Summary

We have seen an improvement for knowledge inference in searching partial information games. The novelty is to include knowledge representation and reasoning into the backtrack partial-information game-tree search. In contrast to *perfect-information Monte-Carlo sampling* used by many AI card playing systems, with a search for a sampled set of worlds, the KBPS search algorithm operates against all possible worlds in one search tree, avoiding the fusion of different card suggestion and resulting in a single card recommendation. It progresses the knowledge in the search tree in an efficient manner, resulting in an optimal search algorithm that is fast enough to be applied in early stages of the game even after a few cards have been played and especially for declarer play, leads to card suggestions that even experienced humans often do not see. If the analysis succeeds, this *killer card* is forced. If not, other card recommendations like expert rules or end game play apply. Although exemplified for *Skat*, the contribution is general to work for other multi-player card games like *Spades*, *Hearts*, *Tarot*, *Marias*, *Ulti*, or *Bridge*, and likely to other domains.

Factorized card-game solving is an almost universally applicable technique to optimize trick-taking play in cases, where the remaining hands among the player, is either known or can be enumerated. It is based on projecting a game to its suits and to combine the result of the according game factorization. It is applied for either the declarer or the opponents to optimize the number of tricks/points for one player, assuming the game is not already won in the current trick. The declarer often optimizes the trump game, since after one trump trick

static information on average distribution and probabilities. is no longer available. Moreover, the declarer can use it for a dynamic computation of standing cards. Similarly, for the opponent to optimize trump and non-trump play for optimizing the number of tricks and points.

We have seen an approach of minigame searches as a widespread almost universally applicable tool to improve card selection in Skat. Although much information about all the other cards in the game is neglected and given that we only approximately combine the partial exploration results directly to an evaluation of the overall search space compared to combinatorial game theory, the factorized approach of optimally solving partial problems helps in many cases for choosing the best card within a suit, once this has been selected.

We have also introduced a rating system for the game of Skat to rank human and automated players for their performance on a longer timescale of play. It includes both opponent strength and card luck and aligns to the accepted scoring system used by the IPSA and DSKV in official tournaments.

A transparent and fair rating system for Skat and other games of chance was wanted to award players in tournaments and online play, setting up live high-score and ranking lists. We have started with a rather simple proposal for computing the ranking that overcomes deficiencies of the existing ELO system for two-player zero sum games that is based on predicting the outcome based on the difference in ELO rating. Like the established ELO system, it is intuitive and can be parameterized. For Skat it offers a fairer evaluation and ranking than the usual approach of simply taking the average score. The system has several advantages: It is comparable to the extended Seeger scoring system used over decades by DSKV/ISPA We reduced the number of series a *fixpoint* for the strength of a player is reached. Using the evaluation of reduced luck, while taking into account opponent strength, we reached a similar fixpoint after a few series. The ELO system itself is deterministic, the same set of games leads to the same ELO value. Different to other researchers looking at ELO evolution to measure chance and skill in the game, the driving force of our research was to counterbalance the chance to converge faster to the real playing strength. By the law of large numbers, one would expect that the more games are played the less important the influence of chance will be in the ELO rating. Besides the factors for chance, we saw this also depends on the volatility, namely factor K.

Besides the Seeger scoring system as its base, there is not much in the proposal that is specific to Skat or to card game, so that we our ELO system can be used to evaluate competitions in AI for many other games of chance and partial information. One simply needs any existing and establish scoring system for a series of games. To reduce the factor for luck, a function estimating the strength of the initial position is needed for normalization. It might be learnt.

13.14 Bibliographic Notes

Skat has been studied in many books [437, 438, 653, 302, 399, 525, 320]. A recent mathematical introduction to Skat playing has been given by [300]. There are frequent bachelor and master theses on the topic of Skat (e.g., by Fabian Knorr, 2018, University Passau, or by Dennis Bartschat, 2019, University of Koblenz), but due to the limited time for programming, the developed bots do not reach a human-adequate playing strength.

Kupferschmid and Helmert [428, 429] developed the *double-dummy Skat solver* (DDSS), a fast open card Skat game solver, which was reimplemented in the *Kermit* player [94]. DDSS was extended to cover partial observable game play using Monte-Carlo sampling [294]. It reached moderate performance results, mainly due to the lack knowledge information exchange between the players.

There have been many efforts to apply machine learning to predict bidding options and hand cards in Skat [396, 395, 94, 583, 530, 531, 273]. Additionally, we have seen feature extraction in the related game of *Hearts* [602], and automated bidding improvements in the game of *Spades* [133]. The results show that prediction accuracy can be improved. Different computer bidding strategies have been proposed in the literature; among others we find neural networks [429], nearest neighbor search [396], and statistical analyses of human games together with single-agent game tree search [456].

Buro et al. [94] indicated that their player Kermit achieved expert-playing strength. A direct comparison with the above AI is difficult, as the bots play on different server architectures.

Cohensius et al. [133] elaborated on an intuitive way of statistically sampling the belief space of hands (worlds) based on the knowledge inferred within play. The matrices P^i for the belief of card location for each player i show a probability $p^i_{j,k}$ for the other players j on having a card k in his hand. While the approach has been developed for *Spades*, it also applies to Skat [584].

In a different line of research, Edelkamp [183] showed how to predict winning probabilities for the early stages of the game, and how to play Null games. Edelkamp [185]) studied Skat endgame play using a complete analysis of the belief-space that is compactly kept and updated in knowledge vectors. Referring to combinatorial game theory [45], Edelkamp [184] proposed suit factorization and minigame search for improved middlegame play in Skat.

Sturtevant and Korf [601] described a paranoid algorithm for the case of perfect information multi-player games. In that work the player to act is paranoid with respect to the preferences of the other players, assuming that they are in a coalition against the agent. This reduces the multi-player to a 2-player game, such that α-β pruning could be applied. Partial-information α-β paranoid search has been considered by Furtak [273]. The work differs from Sturtevant and Korf's algorithm in that the agent does not have to have perfect information. Moreover, because the agent does not know the true world, it is also paranoid with respect to the outcome of any stochastic events (chance nodes), namely the actual distribution of any unobserved cards. The information was used for bidding and stored in tables for the $(32!/10!2!) \cdot 3 \cdot 5 \approx 224$ million hands (including game type and turn), symmetry-reduced and compressed.

Edelkamp [185] considered a similar paranoia partial information search option for analyzing Skat puzzles mainly as a motivation to introduce knowledge representation and reasoning in bitvector sets for endgame play. The algorithm never went into the players' bidding stage, as it was to slow to be useful for the first card under real-world playing constraints.

The general approach to solve a card game with randomness in the deal and partial information is to compute approximate Nash equilibria e.g., by using *counterfactual regret minimization* [76]. As this computation appears not to be infeasible within the given time limits to play a card, approximations have to be found – of which inference, sampling, paranoid search, etc. are some examples. We identified two different approaches for the search of playing cards with uncertainty. One is to generate a set of possible (or all) worlds coherent with the generated knowledge, and, then, to merge the result, possibly improved with dominance checks [110]. This is what is done during endgame play [185]. When the set of worlds is statistically Monte-Carlo sampled with respect to the knowledge of the distribution bias can be given to the distribution. However, the approach often misses the best playing card in early stages of the game, when less knowledge is available. The number of declarer cards unknown to the opponents is important.

Furtak first describes paranoid search for constructing static hand databases [273]. Edelkamp [185] presented another attempt for conducting a search for a forced win against all odds. While interesting for solving Skat puzzles in newspapers, the running time for such early analysis, however, was too large for steady online play. Instead, he successfully integrate this analysis option into actual game play, leading to a considerable increase in playing strength.

Gaming is a multi-billion-dollar industry [142], and games of chance (in contrast to games of skill) are often prohibited, or at least tightly regulated in many jurisdictions over the globe. Thus, the question, whether a game predominantly depends on skill or chance, has important legal and regulatory implications [96, 153, 254]. Öchsler, Dürsch and Lambrecht [172] suggest a new empirical criterion for distinguishing games of skill from games of chance. All players are ranked according to a *best-fit* ELO algorithm. The wider the distribution of player ratings are in a game, the more important is the role of skill. Most importantly, they provide a new benchmark (50%-Chess) that allows to decide, whether games predominantly depend on chance, as this criterion is often used by courts. They have applied the method to large datasets of various games (e.g., Chess, Poker, Backgammon) [169, 97]. The findings indicate that most popular online games, including Poker, are below the threshold of 50% skill and thus depend predominantly on chance. In fact, Poker contains about as much skill

as chess when 75% of the Chess results are replaced by a coin flip. Skat was below der threshold of 50 percent skill, thus depending more on chance.

Duersch, Lambrecht, and Oechsler (2020) studied how many games were affected by chance. They took online playing data and compared the statistics on ELO evolution applying an extension of the Chess formulas. They were not considering the actual input, i.e., for card game the deal. Instead they provide a benchmark test to determine, whether or not a game depends predominantly on chance, through a comparison to a game by randomly replacing 50% of outcomes in Chess with coin flips. In their study by the evolution of ELO values they claimed that 50% Chess still shows more skill elements than Skat, which on this measure for skill versus chance, is on par with Backgammon and above Poker. Even without knowing the game of Skat they found that it is quite sensitive to the factors of card luck (which is undoubted by many players). In Skat, only a considerably small database of played games was taken. Borm and van der Genugten propose measures to compare performances of different types of players. To calculate which part of the performance may be attributed to skill and which to chance, they include as a benchmark an informed hypothetical player who knows exactly which cards will be drawn.

Perfect-information Monte-Carlo sampling (PIMC) introduced by Levy [447] is widely considered to be the best algorithmic options for dealing with such imperfect information games. It has already been used in Ginsberg's popular Bridge-playing program GIB [294], and taken on to other trick-taking games like Skat [429, 273], or Spades/Hearts [602]. An analysis of PIMC has been given by [456]. *Counterfactual regret minimization* [678] is a powerful game-theoretical tool, but the search trees in trick-based card play are widely considered to deep, for its application. For PIMC at each decision point to select a card, it evaluates a larger sample of the belief space and calls a double-dummy solver for each of the worlds, followed by selecting the card with maximum score.

Furtak [273] proposed *recursive Monte-Carlo search* to improve PIMC. Some limitations have been identified for Bridge play as matters of *strategy fusion* and *non-locality* by [110], leading to the $\alpha\mu$ search algorithms. The main observation is that even if the full belief space would be sampled and analyzed, the individual searches in PIMC may lead to contradicting card proposals. The main contribution of $\alpha\mu$ is to increase the lookahead (parameter M) in PIMC for a better exploration/exploitation trade-off. The increase in running time is reduced by further pruning rules. In its nestedness the recursive strategy shares similarities with *nested Monte-Carlo search* [105, 658] and *nested rollout policy adaptation* [545].

Chapter 14
Action Planning

Action planning is an act of *general problem solving*. Given a textual representation of initial state, goal conditions and actions, the task is to find a plan in form of a sequence of action that solves the problem. As action planners are often used as problem solving prototypes, good performance is crucial.

There are two successful approaches to planning, symbolic planning with BDDs to compactly represent and explore sets of states, and explicit-state space search with domain-independent heuristics. The efficiency of heuristic search planning crucially depends on the quality of the domain-independent search heuristic, while a succinct representations of state sets in decision diagrams can save large amounts of memory in the exploration. BDDA* —a symbolic version of A* search with BDDs— combines the two approaches into one algorithm.

We compare two leading heuristics for sequential-optimal planning: the *merge-and-shrink* and *pattern databases*, both of which can be compiled into a vector of BDDs and used in BDDA*. The impact of optimizing the variable ordering is highlighted and experiments on benchmark domains are reported.

Plan recognition is an inference process for generating and narrowing hypotheses. Cost-optimal abduction drives the selection of hypotheses towards the ones with minimal costs. Abduction is inherently complex. We describe solutions for overcoming the computational burden by exploiting a symbolic representation of state sets. Given a model specified in PDDL-like syntax, we infer a discrete variable encoding of the domain and study symbolic algorithms to compute cost-optimal hypotheses and according explanations.

14.1 Introduction

In[1] cost-optimal deterministic action planning, symbolic planners with binary decision diagrams (BDDs) show advantages to explicit-state heuristic search planners, suggesting that the structural savings for representing and exploring large state sets in advanced data structures, sometimes exceed the increased quality of search heuristics.

For the automated construction of search heuristics in BDD-based planning, *symbolic pattern databases* (SPDBs) are the result of a complete (or partial) backward exploration of the concrete (or abstracted) state space. They can be used in symbolic A* search, BDDA* for short.

The *merge-and-shrink* (M&S) heuristic is among the strongest estimates for explicit-state space planning. We extract the M&S heuristic in form of an algebraic decision diagram (ADD). This allows us to enrich a symbolic heuristic planner to exploit this expressive estimate. The precomputed ADD is converted to a vector of BDDs

[1] This chapter is based on joint work with Álvaro Torralba, Vidal Alcázar, Peter Kissmann, Ionut Moraru, Santiago Franco, and Moisés Martínez. It puts together and improves the work from [208, 618, 214, 481].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_14

and plugged into BDDA* for computing cost-optimal plans. It exactly matches the explicit-search M&S implementation and is applicable to all existing variants. We will also look at refinements to BDDA* and propose List BDDA*, which exploits a list representation of the search frontier (rather than a matrix).

SPDBs perform surprisingly well compared to M&S. In the experiments the former computes the perfect heuristic in more instances than the latter. In several cases, the construction does not even need to perform abstraction, but resorts to (possibly truncated) backward search in the concrete state space. While the M&S heuristic is strictly more informed than the PDB heuristic in case of explicit-state search, this is not necessarily true in symbolic search. There are exponential gaps between the M&S and the PDB heuristics that fail to materialize during a symbolic construction. Furthermore, we show that ADD reduction can yield smaller structures than the one applied in the M&S abstraction. We will also see that the variable ordering in the two heuristics is a crucial parameter to the exploration and produces outcomes of large variety.

The most expressive and flexible approach that has been applied to *plan recognition* is abduction. Abduction is the process of finding the cause for a set of assumptions and a theory provided. It is commonly viewed as a form of reasoning, allowing one to find explanations. In literature as well as in this text, the terms *explanation* and *hypothesis* are often used interchangeably. However, we prefer the explanation to refer to the plan generated, and the hypothesis to refer to the possible extensions for the assumptions made.

Of interest is the abductive inference of intended plans. Differently from the plan synthesis problem, in such plan recognition problems, the recognizer is given a fragmented description of the problem and expected to refine it. We concentrate on fully-automated plan recognition in form of plan hypotheses generation wrt a fixed domain theory, a set of observations and a set of assumptions. Supervision via an assisted selection of hypotheses or change in the cost-function is made available. Algorithmically, we apply symbolic abduction, where *symbolic* refers to the use of efficient data structures for representing and operating on Boolean functions.

We also consider modeling and designing algorithms for the abuctive inference in planning problems. We address the issues of computing all valid hypotheses, the uni- and bi-directional inferences of uniform-cost abductions, a setting that is then extended to cover cost-based abductive inferences. User supervision to manually drive the selection of plan hypotheses is discussed next. As the set of abductive inference problems is small, in the experiments we address modified planning benchmarks.

14.2 Symbolic Search

A *planning task* consists of variables of finite domain (so that states are assignments to the variables), an initial state, the goal, and a finite set of operators (each being a pair of preconditions and effects). A well-accepted input formalism is the problem domain description language (PDDL). A simple example of a propositional planning problem domain is given in Programs 14.1 and 14.2.

In *cost-based planning*, operators are associated with action cost values that are integers (rationals can often be scaled to integers). The task is to find a low-cost *plan* from the initial state to the goal. The plan is *optimal* if its cost is smallest among all possible plans.

Program 14.1: PDDL Domain.

```
(define (domain depressed)
  (:requirements :typing)
  (:types person names organ - object)
  (:predicates
     (name ?X - person ?N - names)
     (has ?X - person ?O - organ)
     (like ?X ?Y - person)
     (bad-condition ?Y - person)
     (irreplacable ?Y - person)
     (pessimist ?Y - person)
     (illness ?O - organ)
     (heart-attack ?O - organ)
     (depressed ?X))
  (:action rule-1
  :parameters (?X ?Y - person)
  :precondition (and (like ?X ?Y)
                     (bad-condition ?Y)
                     (irreplacable ?Y))
  :effect (and (depressed ?X)))
  (:action rule-2
   :parameters (?X - person)
   :precondition (and (pessimist ?X))
   :effect (and (depressed ?X)))
  (:action rule-3
  :parameters(?X - person)
  :precondition
   (and (exists (?O - organ)
        (and (has ?X ?O) (illness ?O))))
  :effect
   (and (bad-condition ?X)))
```

Program 14.2: PDDL Problem.

```
(define (problem johnandmary)
(:domain depressed)
(:objects
   j m - person
   john mary - names
   heart - organ)
(:init
   (name j john)
   (name m mary)
   (has m heart)
   (heart-attack heart)
   (like j m)
   (irreplacable m)
   (pessimist j))
(:goal (and (depressed j))
```

Symbolic search uses an *Open list* that contains the states that have been reached but not expanded and a *Closed list* that stores the states that have already been expanded. States are classified by the cost with which they have been reached from the initial state of the search. Therefore, *Open* and *Closed* are lists of BDDs where $Open_i$ and $Closed_i$ represent the sets of states reached or expanded with cost i, respectively. *Closed* is the set of all expanded states. At each step, the algorithm expands the set of states with lowest g-value in *Open* that are not yet in *Closed*, inserting them in *Closed* and all their successors in *Open*.

Bidirectional search performs a forward and a backward searches. The forward search starts at the initial state, and advances towards the goal states. The backward search performs regression from the goal states towards the initial one. The two searches are performed in an interleaved manner so that at each step the algorithm decides whether to continue the backward or forward search. Newly generated states are compared with the set of expanded states from the other search direction. In case of a match, a solution plan has been found, though it is not necessarily optimal. Rather, it is necessary to continue until the plan is proved to be optimal.

In some domains, namely Parking and Tidybot, we the first backward step takes too long, so that the decision whether to use bidirectional or unidirectional BFS could not be finished before the overall time ran out. In these cases, the single images for all the actions were quite fast, but the disjunction took very long. Thus, during the disjunction steps we entered the possibility to check whether too much time has passed. If it has we *stop the disjunctions* and the planner only performs unidirectional BFS. This enabled us to find some solutions in Tidybot. The problem here is that the goal description allows for too many possibilities, because variables from only very few of the finite-domain variable groups are present.

14.3 Heuristic Search Planning

A *heuristic* is a mapping from states to a natural number, and admissible if for all possible states the value is not greater than the cost of an optimal plan. The finite domain variable encoding of the planning problem is often referred to as *SAS*$^+$ *planning*. A planning task *abstraction* is a planning task based on a mapping for the initial state, goal state as well as the operators. We consider two heuristics based on abstraction.

14.3.1 Pattern Database

The pattern database (PDB) heuristic, inspired by a selection of tiles in the sliding-tile puzzle, has been extended to the selection of state variables in other domains and in planning. More general definitions have been applied, shifting the focus from the mere selection of finite-domain variables towards different state-space *abstractions* that are computed prior to the search. A PDB stores the shortest path distance from each abstract state to the set of abstract goal states.

Partial pattern databases truncate backward search at goal distance d, while assigning all remaining states the heuristic value $d+1$. As a slightly better estimate, we can take the minimum value of $i+j>d$ of the goal distance i of a state within the PDB and cost j of an operator.

14.3.2 Merge-and-Shrink

The merge-and-shrink (M&S) heuristic is induced by a *distance-preserving abstraction*. The abstract state space in this heuristic is built incrementally. The rough idea is that finite-domain variables are greedily chosen to construct a larger state space by computing the (synchronous) *product* of the existing state space and the one induced by the next finite-domain variable.

If the state space becomes too large pairs of states are unified (the alternative term *merge* conflicts with the name of the heuristic as the step of merging the states is actually referred to as *splitting*, while the construction of the product state space graph is referred to as *merging*). The approach is layered, so that the union of two state sets is realized by changing the mapping from the state set in one layer to the state set in the next layer. There are different strategies to compute the cross-product state spaces. Most current proposals work on a *linear* arrangement, meaning that one variable is added at a time. Non-linear arrangements that combine arbitrary disjoint variable support sets (limiting the size of the product space) are involved but may yield stronger union operations.

The shrinking is based on the notion of *bisimulation*. Two states s and s' are bisimilar if they are both goal, or every planning operator leads to the same abstract state from both s and s'. If only bisimilar states are aggregated, then M&S is perfect. The bisimulation shrinking strategy computes the coarsest bisimulation, and in the shrinking step it aggregates only bisimilar (abstract) states. In most benchmark domains, however, coarsest bisimulations are still large even under operator projection.

Greedy bisimulation is a relaxed variant of bisimulation, which demands the bisimulation property only for transitions (s,s'), where the abstract goal distance from s' is at most as large as the abstract goal distance from s. This relaxation forfeits the guarantee of providing a perfect estimate.

Motivated by the size of bisimulations, a more approximate shrinking strategy builds the coarsest bisimulation and keeps unifying states until the size limit M is reached. The latter may happen before a bisimulation is obtained, in which case it loses information. The strategy attempts to make errors only in more distant states, where the errors will hopefully not be as relevant.

14.4 Symbolic A* Search

The main limitation for applying PDBs in search practice is the restricted amount of RAM. For the exploration of large state spaces, symbolic search can save huge amounts of memory and computation time. State sets are represented and modified by accessing their characteristic functions.

Decision diagrams are a memory-efficient data structure used to represent Boolean (or integer-valued) functions as well as to perform set-based search, where the diagram represents all binary state vectors that evaluate to certain values. More precisely, a BDD (an ADD) is a directed acyclic graph with one root and two (several) terminal nodes, called sinks. Each internal node corresponds to a binary variable of the state vector and has two successors (low and high), one representing that the current variable is false and the other representing that it is true. For any assignment of the variables on a path from the root to a sink the represented function will be evaluated to the value labeling the sink. Moreover, decision diagrams are unique by applying the two reduction rules of (1) eliminating nodes with the same low and high successors and (2) merging two nodes representing the same variable that share the same low successor as well as the same high successor.

In order to perform symbolic search we need two sets of variables, one (x) representing the current states and another (x') representing the successor states. To find the successors of a set of states S represented in the current state variables given a BDD T for the entire set of actions (i.e., the transition relation) we use the *image* operator, i.e., $image(S,x) = \exists x.S(x) \wedge T(x,x')[x' \leftrightarrow x]$, where $[x' \leftrightarrow x]$ denotes the swap of the two sets of variables. Similarly, we can perform search in backward direction by using the *pre-image* operator, i.e., $pre\text{-}image(S,x') = \exists x'.S(x') \wedge T(x,x')[x \leftrightarrow x']$.

Symbolic PDBs are PDBs that have been constructed symbolically as decision diagrams for later use either in symbolic or explicit heuristic search. Their construction exploits that the transition relation is defined as a relation. The savings observed by the symbolic representation are substantial for many planning domains. Differently from the posterior compression of the PDBs, we work on compressed representation, allowing much larger databases to be constructed. For such *PDB construction*, backward symbolic search is used. In the case of partial PDBs, the construction is truncated at some fixed point in time. While this works in the concrete state space, PDB construction usually takes place in abstract space, imposed by an abstraction function that often projects some variables to don't cares. The automated selection of variables is important for its success but involved.

Algorithmically, we start with the abstract goal set and iterate to successively compute the pre-image. Each state set in a layer is efficiently represented by a corresponding characteristic function. We may assume that the variable ordering is fixed and has been optimized prior to the search. For a given abstraction function the symbolic PDB $Heur(value,x)$ is initialized with the projected goal. As long as there are newly encountered states we take the current backward search frontier and generate the predecessor list with respect to the abstracted transition relation. Then we attach the current BFS level to the new states, merge them with the set of already reached states, and iterate. When action costs are integers this process can be extended from breadth-first to cost-first levels, and it is possible to combine different symbolic heuristics by taking their maximum or by a controlled combination of their sum. The variables encoded in *value* are often queried at the bottom or at the top (in which case we obtain the equivalent to a vector of BDDs). For BDDA* it is more convenient to choose the one where the heuristic relation is partitioned into $Heur_0(x)$, ..., $Heur_k(x)$, with $Heur(value,x) = \bigvee_{i=0}^{k}(value = i) \wedge Heur_i(x)$.

BDDA* operates on a BDD priority queue *Open*. In case of discrete cost-values the *Open* sets can be represented by BDDs. For the organization of the search that avoids BDD arithmetic, it is convenient to partition the state space. As we aim at *cost-optimal symbolic sequential planning*, the matrix-based version of BDDA* works on a partitioning of the search space in g- and h-values, where g is the cost of the path traversed so far and h is the heuristic estimate on the cost to reach the goal. To guarantee optimal cost, BDDA* expands this matrix along the f-diagonals with increasing g-values. The successors of the BDD $Open_{g,h}$ for a chosen transition with cost c are unified with the BDD $Open_{g+c,h'}$, where $h' \in \{0,\ldots,k\}$ is the partitioning obtained by the heuristic evaluation of the successor set.

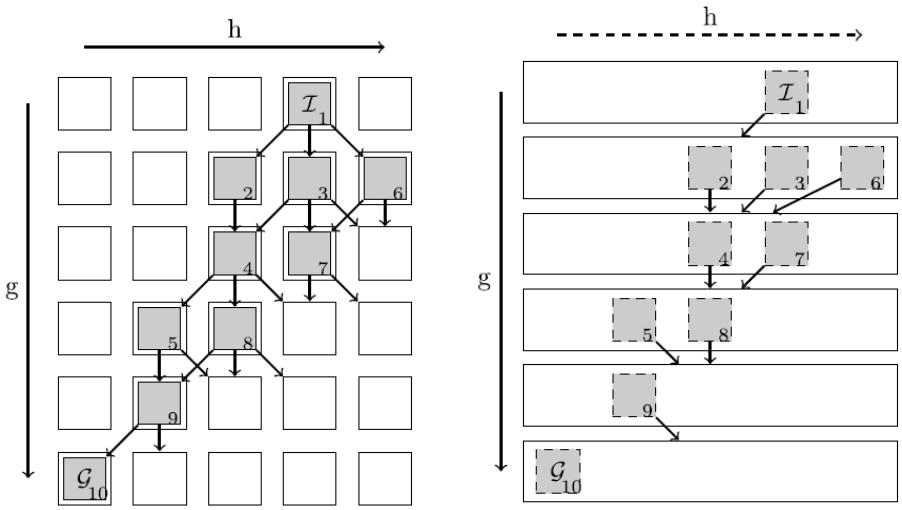


Figure 14.1: Illustration of Matrix BDDA* (left), and List BDDA*, cells are expanded in the order specified by the numbers, arrows denote successor buckets.

14.4.1 Basic Improvements

The starting point for the IPC-7 winning planner SymBA* is the IPC-6 version of the planner Gamer. It applies symbolic PDB construction (15 minutes) and BDDA* search (15 minutes) for cost domains or bidirectional BFS (30 minutes) for unit-cost domains. If backward search takes too long, abstractions are applied, otherwise a (partial) PDB in the concrete search space is constructed. If we compare the number of solved instances of the domains with and without action costs the results are quite peculiar. For the domains without action costs Gamer found only 36 solutions; only one participant was worse than that. For those with action costs Gamer found 112 solutions; only four other planners were able to find more (with the maximum being 120). Based on the results of the competition small improvements were implemented, which are as follows.

In the *solution reconstruction* for bidirectional BFS, Gamer supposed that at least one forward and at least one backward step were performed. The two easiest problems of VisitAll require only a single step, so that the solution reconstruction crashed.

In some cases, *parsing* the ground input took more than 15 minutes, so that actually no search whatsoever was performed in the domains with action costs: At first, it was parsed in order to generate a PDB; this was killed after 15 minutes, and then the input was parsed again for BDDA*. In the domains without action costs it sometimes also dominated the overall runtime.

In the most complex cases, generating the BDDs for the *transition relation* takes a lot of time, as well. The planner had to generate them twice in case of domains with action costs if it did not use the abstraction, once for the PDB generation and once for BDDA*. Instead, we store the transition relation BDDs, the BDD for the initial state and that for the goal condition on the hard disk; reading them on disk is often a lot faster than generating them again from scratch.

While the original planner used bidirectional breadth-first-search (BBFS) for domains without action costs, we tried running *BDDA* in all cases*, no matter if we are confronted with them or not. Thus, for the domains without action costs we treated all actions as if they had a cost of 1. We call this implementation *Matrix BDDA** (see Figure 14.1, left).

Algorithm 14.1: List-BDDA*.

Input: $\mathscr{A}$: finite set of operators, $\mathscr{I}$: initial state, $\mathscr{G}$: goal description.
$c : \mathscr{A} \mapsto \{1,\ldots,C\}$: action costs.
$Heur_h$: heuristic (with h being the maximal heuristic value).
T_a: transition relation for operator $a \in \mathscr{A}$.
Output: cost-optimal plan.
$Open_0 \leftarrow \mathscr{I}$
for all $f = 0,\ldots$
  for all $g = 0,\ldots,f$
    $h \leftarrow f - g$
    $S \leftarrow Open_g \wedge Heur_h$
    if $(h = 0)$ **and** $(S \wedge \mathscr{G} \neq 0)$ **return** *ConstructSolution*
    for all $i = 1,\ldots,C$
      $Succ_i(x') \leftarrow \bigvee_{a\in\mathscr{A},c(a)=i} \exists x \,.\, S(x) \wedge T_a(x,x')$
      $Open_{g+i} \leftarrow Open_{g+i} \vee Succ_i$

*14.4.2 List BDDA**

In Matrix BDDA*, all successors are classified according to their h-value by applying conjunctions with all the heuristic BDDs. When the number of heuristic values grows, this can be inefficient, since some of these conjunctions could be avoided.

The representation in the matrix can be simplified to the vector for the states in the *Open* list ordered along the g-value. The reasoning behind this strategy is to defer the heuristic calculation by computing the conjunction of the successor set with the heuristic estimate only when it is needed for expansion in the currently traversed f-diagonal.

The pseudo-code of the resulting algorithm *List BDDA** (see Figure 14.1, right). for non-zero cost operators is shown in Algorithm 14.1. All inputs to the algorithm $\mathscr{A}, \mathscr{I}, \mathscr{G}, c, Heur_h, T_a$ are represented as BDDs.

It is simple to add duplicate detection to the algorithm using another set *Closed* for the set of expanded states. The handling of zero-cost operators adds another BFS loop to the code as these operators are to be preferred in the exploration. While Matrix BDDA* uses BFS to get the states reachable with zero-cost operators independent of their actual h values, in the list version we apply a conjunction with the heuristic value to get only those states in the current f-diagonal.

14.5 Symbolic Merge-and-Shrink

It is not difficult to observe that the precomputed memory structure of the M&S heuristic can be cast as a symbolic representation of an integer-valued function. This function can be extracted in form of an ADD allowing us to enrich a symbolic heuristic planner to exploit this expressive estimate. The precomputed ADD is converted to a vector of BDDs and can be plugged into an optimal symbolic heuristic search planner.

Every intermediate abstraction corresponds to a layer in the ADD and each abstract state corresponds to an ADD node. When a new variable is merged into the abstraction, every state is split into k states in the next level, one for each value of the variable. ADD nodes representing the parent state are connected with the nodes representing its successors. As M&S works with finite domain variables but the ADD is defined for binary variables, each node with k successors is converted to a binary tree with $\lg k$ layers.

To compute the ADD of an M&S heuristic we start by generating the sink nodes associated with the different heuristic values of the abstract states in the last layer. Then, recursively, nodes in the previous layer can be

constructed pointing to the nodes in layers already computed. During the ADD construction we ensure the application of the reduction rules, so that the size of the final ADD is usually smaller than the M&S heuristic structure.

14.5.1 ADD Complexity

The symbolic ADD representation of the M&S heuristic can be computed in time and space $O(nM)$, where n is the size of the Boolean state vector and M is the pre-defined maximum number of states. Moreover, the representation of the heuristic as a sequence of BDDs $h_0, \ldots, h_{\max}$ can be computed in time and space $O(h_{\max} nM)$. The time and space complexities are implied by the maximum sizes of the state spaces for the construction of the next-variable tables in the explicit search construction of the M&S heuristic. BDD reduction is a linear time operation and only decreases the size.

The ADD sizes for the two M&S heuristics are shown in Table 14.1. For each domain we provide the number of instances in which the heuristic computation was finished in 30 minutes as well as the sizes of the largest ADD for each domain. Surprisingly the ADDs are small, especially for the greedy version of M&S, showing that not much memory is spent once the ADD has been computed.

Table 14.1: Number of instances with M&S heuristic (#) and maximum number of ADD nodes over all instances (n) for all domains of the sequential optimal track of IPC 2011.

	M&S (gop')		M&S (gop)	
Problem	#	n	#	n
Barman	20	177,294	20	45
Floortile	20	1,278,950	8	6,283
NoMystery	20	197,445	20	915
Parking	0	—	20	2,260
Tidybot	0	—	20	15
VisitAll	20	3,225,813	20	7,381
Elevators	20	62,594	0	—
Openstacks	20	102,486	4	134,780
PARC-Printer	19	4,606,533	20	11,788
Peg-Solitaire	20	42,170	0	—
Scanalyzer	16	356,698	6	29,921
Sokoban	20	1,339	1	33,525
Transport	20	257,898	20	753
Woodworking	20	248,263	20	439,489

14.5.2 Limits and Possibilities

In explicit search, the M&S heuristic strictly generalizes the PDB heuristic, as (with only merging) by computing their synchronous product all pattern database heuristics based on projecting the variables can be constructed. In some cases M&S can compute perfect heuristics in polynomial time, where PDBs cannot. The distinguishing example is the Gripper domain.

In a symbolic setting, this reasoning, however, is no longer immediate. If all the variables are included in the pattern, the original state space can be fully traversed resulting in the optimal heuristic.

And indeed, the BDD exploration that computes the perfect heuristic in Gripper is polynomial. As the representational power of both alternatives is equivalent (an ADD for M&S and a list of BDDs for the symbolic PDB) both approaches can potentially derive optimal heuristics in the same domains.

However, even if the M&S bisimulation gets the perfect heuristic, it does not always result in a reduced representation of the ADD. First, we observe that for any (e.g., the perfect) heuristic – no matter how it is computed – by the uniqueness property, the according ADDs (following the same variable ordering) have to be the same. Secondly, we can construct an intuitive example, where shrinking is not able to compute the most reduced form of the heuristic.

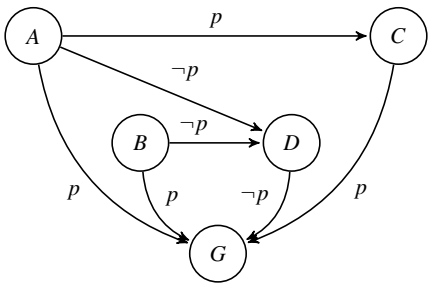


Figure 14.2: Example of bisimulation. A, B, C, D and G are states in one level of the M&S process, while p and $\neg p$ are variable assignments that serve as a precondition of the according operators.

Figure 14.2 shows an example where there are not any bisimilar states. The transition labels have already been reduced so that they refer to variables that have not yet been merged. In the example these labels are preconditions and they only refer to a binary variable p. All the transitions have unitary cost and the goal is to reach state G.

We say that two abstract states s and s' are *equivalent* if and only if, for every value assignment to the variables that have not yet been merged the goal distance remains the same. If two abstract states are equivalent, their corresponding ADD nodes can be unified according to the ADD reduction rule (2). It is easy to see that states A and B in the example are equivalent because in case that p holds both have a cost of 1, while if $\neg p$ holds both have a cost of 2. However, they are not bisimilar because B does not have any transition to state C. Obviously, states C and D are not bisimilar, given that their transitions have different labels. Therefore, no pair of states is reduced by bisimulation.

However, since in the end only the distance to the goal matters, those transitions that are not part of an optimal path should not be taken into account. In the example, if the transition $A \rightarrow C$ is not necessary then A and B are equivalent. Checking if a transition is necessary in any optimal path is not trivial as it needs to consider all the combinations of values of the variables that have not been merged.

It is possible to extend the example by adding an exponential number of equivalent states that are not bisimilar because they have different transitions that are not needed by any of their optimal paths. Therefore, this can cause an exponential gap between the size of the intermediate abstraction and the final reduced heuristic.

On the other hand, symbolic backward search iteratively constructs the reduced BDDs for every cost. In the absence of 0-cost actions, the intermediate BDDs are always fully reduced. Thus, in some domains the size of the BDDs used by symbolic partial PDBs may be exponentially smaller than the M&S representation. The counterpart is that these BDDs are computed with images of the transition relation, which in some domains may be expensive.

14.6 Comparison

We use two planners as the basis for the first experiments, namely Fast Downward (FD), offering the M&S heuristic for explicit-state planning, and Gamer for executing symbolic heuristic search.

The software infrastructure is taken from the resources of the International Planning Competition. While in the 6th edition the BDD-based planner Gamer won the sequential optimal track, in the 7th edition explicit-state heuristic search planners, especially portfolio planners took the lead. For the 8th edition, we implemented the refinements in Gamer (Matrix BDDA* and List BDDA*) using the CUDD library of Fabio Somenzi. Time and memory settings are the same as in the competition, other parameters of the computer are slightly different.

We experimented with two different configurations of the M&S heuristic, one using only greedy bisimulation (gop) and one using DFP-gop (gop'). The M&S planner presented in the competition used these two strategies serially, first the version with gop and afterward the one with DFP-gop. We decided to run both parts independently to see how well we perform against a more traditional, i.e., non-portfolio, planner as well. The pattern selection for symbolic PDBs is the same as that used by Gamer.

We look at two different variable ordering strategies used by the competition planners, the one applied in FD and the one applied in Gamer. The FastDownward ordering looks at strongly connected components and weights of the *causal graph*. Highly related variables are pushed to the top and goal variables are pushed to the bottom of the ordering. The Gamer ordering also looks at the dependency of variables and is the result of a random local search to improve the ordering by incrementally computing the optimization function $\sum_{1\leq i,j,\leq n,(u_i,v_j)\in D}(\pi(i)-\pi(j))^2$, where π denotes the applied permutation and D denotes the set of the causal dependencies. Thus, highly related variables are pushed to the middle of the ordering.

Table 14.2: Number of solved problems for all domains of the sequential optimal track of IPC 2011.

Domain	FastDownward Ordering							Gamer Ordering							
	Explicit A*		Matrix BDDA*			List BDDA*		Explicit A*		Matrix BDDA*			List BDDA*		
	gop	gop'	gop	gop'	PDB	gop	gop'	gop	gop'	gop	gop'	PDB	gop	gop'	PDB
Barman	4	4	4	4	4	4	4	4	4	6	5	**8**	4	4	4
Floortile	3	7	3	7	**12**	3	7	3	4	3	4	9	3	4	8
NoMystery	13	**20**	14	**20**	16	15	**20**	13	12	14	17	14	13	18	13
Parking	**7**	0	3	0	0	3	0	0	0	0	0	1	0	0	1
Tidybot	**13**	0	6	0	6	6	0	**13**	0	9	0	8	6	0	5
VisitAll	**13**	11	5	5	12	12	12	11	9	11	10	11	11	10	11
Total (no-cost)	**53**	42	35	36	50	43	43	44	29	43	36	51	37	36	42
Elevators	0	11	0	16	**19**	0	16	6	12	6	14	**19**	5	17	**19**
Openstacks	4	16	4	15	14	4	15	5	16	4	**20**	**20**	5	**20**	**20**
PARC-Printer	11	**14**	8	11	7	10	11	11	12	8	9	7	10	7	8
Peg-Solitaire	0	**19**	0	**19**	**19**	0	**19**	0	**19**	0	**19**	17	0	**19**	17
Scanalyzer	6	**10**	6	9	9	6	9	3	8	3	8	9	3	7	9
Sokoban	1	**20**	1	13	12	1	12	3	**20**	2	18	19	2	18	19
Transport	6	7	6	7	**9**	6	7	6	6	6	6	7	6	6	8
Woodworking	9	6	6	5	5	10	7	9	9	6	8	**16**	13	12	**16**
Total (cost)	37	103	31	95	94	37	96	43	102	35	102	114	44	106	**116**
Total (all)	90	145	66	131	144	80	139	87	131	78	138	**165**	81	142	158

The results are shown in Table 14.2. All the small improvements in Matrix A* compared to Gamer helped mainly in the domains without action costs. There we are now able to find the two trivial solutions in VisitAll, in Barman we find eight solutions – twice as many as anyone else in the competition – in Parking we find one solution and in Tidybot we increase the result to eight solutions. In the domains with action costs the new parser helped us to find three additional solutions in the Scanalyzer domain. Overall, Matrix BDDA* solves 165 problems, which is 19 problems more than the competition version of Gamer.

When comparing both implementations of BDDA*, List BDDA* improves the results when the M&S heuristic is used but is worse than Matrix BDDA* with symbolic PDBs. As this happens only for unit-cost domains, we have to further investigate its cause. It is possible that the BDD representation of all states that have been visited becomes too big or that timing issues prevent the algorithm from solving the six problems. On the other hand, explicit A* beats both BDDA* versions when using FastDownward ordering, but with Gamer ordering the results are slightly better for BDDA*.

The symbolic PDB heuristic did not use abstraction in most of the domains. With Gamer ordering abstraction is used in some problems of PARC-Printer, Parking, Sokoban and Tidybot. With FastDownward ordering abstraction is also used in Floortile and Openstacks. In all the other domains the heuristics were computed by symbolic backward search until the all states had been reached or the time limit of 15 minutes had been expired. The perfect heuristic was found for 70 problems when using Gamer ordering and for 60 with FastDownward ordering.

The results are highly influenced by the choice of the variable ordering. Overall the FastDownward ordering is better for the M&S heuristic, while Gamer's ordering helps the symbolic exploration. Due to this, the integration of symbolic search and the M&S heuristic is difficult because both have to use the same ordering.

The variable ordering matters not only for the kind of planner used but also for the domain it is used on. For example, in Floortile, NoMystery, Parking, VisitAll, PARC-Printer, and Scanalyzer the FastDownward ordering is better in most cases for all planners, while in Elevators, Openstacks, Sokoban, and Woodworking the Gamer ordering is the better choice.

Overall, the best single planner is Matrix BDDA* using symbolic PDBs with the gamer ordering. However, as the M&S heuristic takes less time, it is more suitable to be run more than once. Using the same configuration as in the competition, FD with the M&S heuristic can solve 171 problems, two more than in the competition, probably due to some bug fixes and performance boosts in that planner as well. Neither of the two versions took more than 600s for any of the problems, so that a combination of both really is reasonable. The memory limit of 6 GB is what prevents them from finding more solutions.

The 8th IPC included a temporal, a sequential, and an agile track, optimizing different criteria for plans (action cost, CPU time and total-time). The PDDL input specification language has not been changed, but more emphasis was given to conditional effects. Probably most exiting was the cost-optimal sequential planning track, enforcing plans of minimal sum of action costs (see Figure 14.3). The top five performing planners were SymBA*-2 (151 of 280 problems solved), SymBA*-1 (143), cGamer (120), SPM&S (114), RIDA* (113), and Dynamic-Gamer (99). All but one of these systems exploit (reduced ordered) binary decision diagrams (BDDs) for state-space traversal and heuristic guidance.

14.7 Complementary

The Complementary planner (1 & 2) uses pattern databases (PDBs). Recall, that a PDB is a heuristic function in the form of a lookup table that contains optimal solution costs of a simplified version of the task. The planner dynamically creates multiple PDBs, which are later combined into a single heuristic function. At a given iteration, the method uses estimates of the search space size to create a PDB that complements the strengths of the PDBs created in previous iterations.

The biggest difference to earlier work, which always started with smaller PDBs and used *a priori* time limits to sequentially increase the PDB's size limit, the applied method has no such schedule nor an initial bias. Complementary uses the UCB1 bandit algorithm to learn which PDB sizes fits best the current problem given the previously selected PDBs. Moreover, two seeding algorithms based on bin packing are studied, and we also added a pattern generation algorithm based on Gamer. Finally, the code itself has been refactorized to ease the addition of evaluation methods, generation methods and other alternative configurations.

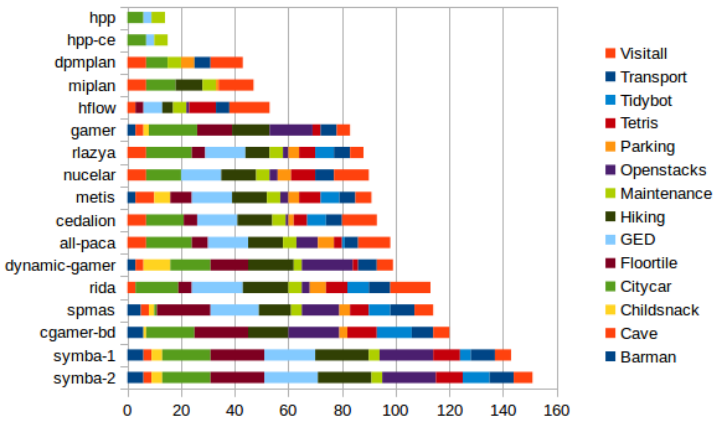


Figure 14.3: IPC-8 results.

CPC sequentially creates a set of pattern collections $\mathscr{P}_{sel}$ for a given planning task ∇. Regular CPC starts with an empty $\mathscr{P}_{sel}$ set and iteratively adds a pattern collection $\mathscr{P}$ to $\mathscr{P}_{sel}$ if it predicts that $\mathscr{P}$ will be *complementary* to $\mathscr{P}_{sel}$. We say that $\mathscr{P}$ complements $\mathscr{P}_{sel}$ if A* using a heuristic built from $\mathscr{P} \cup \mathscr{P}_{sel}$ explores a smaller search space than when using a heuristic built from $\mathscr{P}_{sel}$. CPC uses estimates of A*'s search space to guide a local search in the space of pattern collections. After $\mathscr{P}_{sel}$ has been constructed, all the corresponding PDBs are combined with the canonical heuristic function.

An evaluation of the pattern selection scheme in different settings including explicit and symbolic PDBs showed that combining symbolic PDB heuristics were able to outperform existing methods. Futhermore, it also showed that CPC could create complementary PDBs to other methods. The best combination was using the method to complement a symbolic perimeter PDB. The selected method to be complemented for this competition first generates a symbolic PDB up to a time and memory limit. A maximum amount of BDD nodes in the perimeter frontier was also used as a failsafe on the actual implementation (otherwise the code occasionally would get stuck while generating the next step for the BDD generation). One advantage of seeding the algorithm with such a perimeter search is that if there is an easy solution to be found in what is basically a brute-force backwards search, we are finished before we even start finding complementary PDBs. If a PDB contains all available variables, any optimal solution for such abstraction is also necessarily an optimal solution in the real search space. In such cases we stop building the perimeter and simply return the optimal plan found.

For this planner, we have added two new seeding methods besides the perimeter PDB we collectively refer to as *bin-packing*. The first one uses *first-fit increasing* (FFI) to try to find the smallest collection of PDB using the bin packing principle. The second method uses *first-fit decreasing* (FFD) to do the same. Bin packing for PDBs tries to create the smallest number of PDBs which uses all available variables. While reducing the number of PDBs used to group all possible variables does not guarantee a better PDB, the less number of collections, the less likely on average to miss interactions between variables due to being placed on different PDBs (The packing algorithm used here ensures that each PDB has a least one goal variable and also that all variables in a PDB are casually connected, on their own or through a chain of local variables, to at least one goal variable in the PDB). PDB selection methods tend to suffer from diminishing returns, i.e., the more time invested using a pattern generation method, the less likely it is to find a new improving one. Using different PDB generation methods or varying their parameters, e.g. PDB size limits, is how we try to ameliorate diminishing returns.

If no solution is found after the perimeter PDB is finished, the method will start generating pattern collections stochastically until either the generation time limit or the overall PDB memory limit are reached. CPC decides whether to add a pattern collection to the list of selected patterns if it is estimated that adding such PDB will speed up search. We used the stratified selection time prediction method to estimate this. When a pattern collection is added, all its patterns are collected using the canonical combination method.

We compared the pattern selection methods to the Gamer algorithm. Gamer is based on the idea of trying to discover the single best possible PDB for a problem. Its pattern selection method can be summarized as an iterative process, starting with all the goal variables in one pattern, where the casually connected variables who would increase the most the average h value of the associated PDB are added to the Gamer pattern. We have created a new *Gamer-style* pattern generation method which behaves similarly, more details on the next section. This pattern selection method is intended to be complementary to the ones in the original CPC methods.

Once all patterns have been selected, the resulting canonical PDB combination is used as an admissible heuristic in A* search for the sequential optimal track. A cost-bounded option was added, in form of a slightly modified version of lazy greedy search. The modification is that instead of pruning all generated successor nodes whose g value is above the bounded cost, we actually prune all nodes whose g+h values are above the bounded cost. This is only guaranteed to keep solution cost at or below the bounded cost if the heuristic is admissible. Since this is the case for the heuristic, there is no reason to take advantage of this.

This planner was not submitted for the satisficing planninng track due to the inherent incompatibility of the heuristic and the track criteria. Generating large symbolic PDBs cost a significant amount of time. Finding which patterns make good pattern collections is even more costly. In satisficing planning, the critical factor is finding a solution as quickly as possible, and, hence, it is generally better using heuristics with small preprocessing costs.

14.7.1 Configuration Choices

For the 8th IPC in Complementary (1) the following list of changes to the base planner were made.

- Moving to a 64 bits build, due to the increase of memory limit on the IPC. This required doubling the relevant PDB and overall memory limits.
- Using only symbolic (BDD-based, no explicit-state) PDBs.
- After the initial perimeter search was finished, two different bin-packing algorithms were run to generate an optimized variable distribution to generate the PDBs: a) first-fit decreasing with a time limit of 50s distributes the variables in different bins according to their size in bits (variables are initially shorted by their size, then, the smaller variables are grouped in the first bins, while the bigger are grouped in the last ones and sometimes on their own); b) first-fit increasing with a time limit of 75s distributes the variables in different bins according to their size as the previous one but in this case the bigger variables are grouped in the first bins while the smaller are grouped in the last ones. Empirical tests have shown first-fit increasing to perform better on average.
- dropping stratified sampling, as for symbolic PDBs performance the method was no longer better.
- The planner applies the UCB1 algorithm to learn *in situ* which PDB size limits are likeliest to perfrom better; UCB1 will change the recommended PDB size limits if diminishing returns become a significant problem for a specific PDB size bracket. On the original version, fixed time limits were given to increase the PDB size limit by an order of magnitude, potentially forcing the heuristic to keep trying a size limit not justified by the problem data.
- The modified CPC algorithm decides on each iteration which pattern generation method to use. We use UCB1 also to learn, whether to use the CBP generation method or the Gamer-inspired method. The Gamer algorithm has a termination condition if no variable can be added to improve the average heuristic value of the selected pattern. In this case, calling the Gamer generation method was stopped if no variable could sufficiently increase the average heuristic value given the current time and size limits.
- The other pattern generation methods start from scratch, however, for the Gamer style pattern selection method, the choice is always whether to add variables to the previously selected pattern. For the Gamer-inspired pattern generation, we use the average heuristic values to decide whether the next iteration is

improving the pattern. If no variable is found that sufficiently increases the average heuristic value of the Gamer-style pattern, this method is dropped from the available pattern selection methods UCB1 can select from. However, in terms of comparing the Gamer style pattern with the already selected patterns by CPC we use the *in situ* probing mechanism based on problem data, in this case whether the size of the search space is predicted to be reduced by adding the new Gamer style pattern.

- UCB1 has also been used to decide how many goal variables are present in a single pattern, while the original CBP method was seeded by just one goal variable per pattern. We noticed that one of the reasons Gamer does so well for some problems is that it starts with all the goal variables. For some problems, missing even one goal variable in each pattern when using CBP has resulted in much lower accuracy. UCB1 learns if this is the case in the current problem. As an added bonus, it increases the diversity of PDB generation methods and, hence, hopefully ameliorate diminishing returns.

14.7.2 Results

We performed an ablation-type study to analyze which components worked best (Table 14.3, domain names have been abbreviated).

Table 14.3: Coverage of Complementary1 Modules on the IPC-9 benchmark. Reg stands for all components active. "NoPer" stands for perimeter PDB inactivated. "+BinPack" stands for using PDBs generated by bin packing generator. "CBP", "Cgamer" and "BinPackOnly" rows also have Perimeter inactive.

Domain/Method	Agr	Cal	DN	Nur	OSS	PNA	Set	Sna	Spi	Ter	Total
Comp1/Reg	10	11	13	12	**12**	**19**	**9**	10	**12**	**16**	124
Comp1/NoPer+BinPack	10	**12**	**14**	**14**	**12**	**16**	**9**	12	11	**16**	126
Comp1/NoBinPack	6	11	13	**14**	**12**	**19**	**9**	11	11	**16**	122
Comp1/CBP+BinPack	8	**12**	**14**	13	**12**	**18**	**9**	11	11	**16**	124
Comp1/CBP-NoBinPack	6	**12**	**14**	13	**12**	18	**9**	9	11	**16**	120
Comp1/Gamer+BinPack	**13**	**12**	**14**	**14**	**12**	17	**9**	12	11	**16**	**130**
Comp1/Gamer-BinPack	8	12	12	**16**	**12**	18	**9**	**14**	11	**16**	128
Comp1/BinPackOnly	7	**12**	14	12	**12**	7	**9**	11	11	12	107
Solved by any of the Comp1 methods above											
*	14	12	14	16	12	20	9	14	12	16	139
Competition result below included for Completeness											
Comp1	10	11	14	**13**	13	17	8	11	**11**	**16**	124

In the competition, Complementary1 solved the same number of overall problems as Complementary2, however, their ablation studies are different. Table 14.3 shows the results with (Comp1/Reg) and without the initial perimeter PDB (Comp1/NoPer), with and without the seeding bin packing generator and finally for each of the individual packing method on their own (CBP, Gamer).

The biggest difference with Complementary2 is that dropping the initial perimeter PDB would have increased the number of overall solved problems by 2 and, hence, would have tied with the competition winner in terms of problems solved.

It was also surprising that the Gamer module, seeded by BinPack (Comp1/Gamer+BinPack), solved 4 more problems than if we use the selection mechanism (Comp1/NoPer+BinPack) and would have actually won the competition. This indicates that for some problems the best option was to keep growing the Gamer-style pattern but instead CBP was selected (or we run out of time before we could grow the Gamer-style PDB to the same size).

Finally, we also included the best possible results if we knew the best pattern generator method *a priori* for free: 139 problems are solvable using the right combination of generation methods, 15 more than with the chosen

selection mechanism. Of course, this comparison is biased, i.e., when running the selection mechanism the whole available time has to be split between the preferred pattern generators, while when picking the best out of each methods, each of them had the whole generation time. This means the number of patterns tested is much larger. It indicates, however, the potential of symbolic PDBs still have in competitive planning.

In the 8th competition the Complementary systems ended up as runners up, the winner solved 126 problems while it solved 124. It was the best non-portfolio approach. A Gamer-style approach would have resulted in the best overall results. This confirms that choosing the Pattern Generator is very much a question of which domain is it to be used for.

Interestingly, when running each pattern generation method on its own, 15 more problems were solvable. Using the two bin-packing generators to seed the heuristic proved useful. It improved the overall results for all the methods we tested, even though the implementation for bin packing did not include any stochastic method. Generally, the more diverse the pattern generators are, the more likely it is for one of them to find good patterns. The preliminary results here seem to indicate that using bin packing techniques as pattern generators is qpromising as a complement to CBP and Gamer.

Finally, it seems that using the UCB1 bandit algorithm to learn which PDB sizes fit the current problem best, given the previously selected PDBs, has resulted in lowering the dependency on the perimeter PDB to obtain best results, whereas in the old implementation dropping the perimeter results in solving 10 fewer problems. Moreover, the bandit version solves more problems without the perimeter PDB, regardless of the combination of pattern generators used.

14.8 Symbolic Abduction

In abduction, for a logical theory T and some manifestation M of a set of individual hypotheses, we are interested in Δ such that $T \cup \Delta \models M$.

Driven by the success of recent BDD-based planning systems on international planning competitions the large BDD compression ratios for many planning benchmarks, we aim at solving the abduction problem with BDDs by embedding it into the planning domain definition language PDDL.

PDDL problems can be grounded using static analyzers that instantiate predicates, actions and fluents with all possible instantiations of domain objects, yielding a (usually fully instantiated) initial state I, a set of operators O, and a goal description G. Despite its binary representation for symbolic search, it is best to consider all states and operators as sets. For example, set intersection matches Boolean conjunction, set complementation matches Boolean negation, and set unification matches Boolean disjunction.

For abduction, we assume the domain theory T to be encoded in the planning operators O. In BDD terminology, we construct a transition relation T_o, encoding all (predecessor, successor) state pairs valid under operator $o \in O$. This yields the domain theory $T = \bigvee_{o \in O} T_o$. Logical subsumption $\phi \models_T \psi$ inherits the semantics that there is a sequence of operators applied to ϕ, which entails ψ.

For abductive inferences, again we assume a finite-domain variable planning task structure that static analyzers in many planners induce as the basis. The information on mutual exclusion that is encoded in the finite domain variable description belongs to the set of consistency condition provided to the abductive inference module. In other words, we assume the manifestation to be separated in the set of observations G in form of a partial description of the goal state, and the set of assumptions A in form of a partial description of the initial state. We are interested in some hypothesis Δ, such that $A \cup \Delta \models_T G$.

Any aductive inference process partitions in two stages: (1) generating all, a subset of them, or only one hypothesis, and (2) selecting the hypothesis that is best, which can be either automated wrt some optimality criterion, or interactive by modifying the assumptions or the observations, in which case the abductive inference iterates.

14.8.1 Computing Valid Hypotheses

The ultimate goal is to compute all valid hypotheses. Eiter and Makino generate all nontrivial explanations of a Horn-CNF wrt some positive letter. For each generated hypothesis the algorithm is polynomial, but the number of hypotheses can be exponential.

To compute all possible hypotheses Δ, we first define the preimage of a state set *States* (on variable set x') as

$$PreImg(States) = \bigvee_{o \in O} \left(\exists x'.T_o(x,x') \wedge States(x')\right)[x \leftrightarrow x'].$$

The suffix $[x \leftrightarrow x']$ denotes that the state variable set is swapped after the operation. If $States_i$ denotes the representation of the set of states in some backward breadth-first search (BFS) level i (minimal goal distance i), then $PreImg(States_i)$ denotes the set of states in backward BFS level $i+1$. Moreover, the set of all states that are reachable via pre-image is defined as

$$BackReach(States) = \mu X.PreImg(X) \vee States(x'),$$

where μ denotes the fixpoint operator induced by repeated pre-image application. When initializing *States* with goal condition G, with the above equation we compute all possible states that reach G, i.e., *BackReach* is partitioned in BFS levels $BackReach_i$, with $BackReach_0 = G$, and $BackReach_{i+1}$, is computed from $BackReach_i$. To guarantee termination of the exploration, it is recomended to subtract $BackReach_j$ from $BackReach_{i+1}$ for $0 \leq j \leq i$.

For a set of assumptions encoded as a formula $A(x)$ the set of all valid hypotheses is now computed as

$$ValidHypotheses(G(x)) = BackReach(G(x)) \wedge A(x),$$

i.e., the set of all possible states that can reach the observations and that satisfy the assumptions.

14.8.2 Uniform-Cost Abductive Inference

In the case of uniform-cost abductive inference, we follow the principle of Occam's razor to compute the step-minimal explanation. As an example, we take the case of John being depressed under the condition that his girl-friend Mary has had a heart attack. The step-minimal explanation for him being depressed is that John is a pessimistic person, but this is commonly interpreted as the unlikely explanation, given the evidence of Mary's illness.

Backward uniform-cost abduction is shown in Algorithm 14.2. It repeatedly applies preimages until the assumptions A are hit, inducing a plan to be generated (for the sake of simplicity, the test of termination for inconsistent assumptions is not shown. It requires the maintenance of a closed list and leads to a full backward exploration of the state space). In $Min \wedge A$ there might be several valid minimum-step hypotheses. If A' is one, then Δ is the completion to the set of assumptions already made.

As a feature of the algorithm, a prediction Γ to the set of observations can also be returned, by completing the partial goal G to the complete one G' as found through the construction of the explanation.

The advantage of BDD inference is that it is easy to invert the direction of chaining and compute the image of some state set *States* as follows:

$$Image(States) = \bigvee_{o \in O} (\exists x.T_o(x,x') \wedge States(x))[x' \leftrightarrow x].$$

Algorithm 14.2: Backward abduction on uniform cost problems.

Procedure Backward BDD-Abduction
Input: Uniform cost planning problem with theory $T = \bigvee_{o \in O} T_o$, set of assumptions $A \subseteq AP$, and set of observations G
Output: Step-optimal explanation Δ that is consistent with $A \wedge \Delta \models_T G$

$BackReach_0(x) \leftarrow G(x)$
for each $i = 0, 1, \ldots$
  $Min(x) \leftarrow BackReach_i(x)$
  if $(Min(x) \wedge A(x) \neq false)$
    return $ConstructExplanation(Min(x) \wedge A(x))$
  $Pred_l(x') \leftarrow \bigvee_{o \in O}(\exists x'(Min(x') \wedge T_o(x, x')))[x \leftrightarrow x']$
  $BackReach_{i+1}(x) \leftarrow BackReach_{i+1}(x) \vee Pred_l(x)$

Algorithm 14.3: Forward abduction on uniform cost problems.

Procedure Forward BDD-Abduction
Input: Uniform cost planning problem with theory $T = \bigvee_{o \in O} T_o$, set of assumptions $A \subseteq AP$, and set of observations G
Output: Step-optimal explanation Δ that is consistent with $A \wedge \Delta \models_T G$

$ForwReach_0(x) \leftarrow A(x)$
for each $i = 0, 1, \ldots$
  $Min(x) \leftarrow ForwReach_i(x)$
  if $(Min(x) \wedge G(x) \neq false)$
    return $ConstructExplanation(Min(x) \wedge G(x))$
  $Succ_l(x) \leftarrow \bigvee_{o \in O}(\exists x(Min(x) \wedge T_o(x, x')))[x \leftrightarrow x']$
  $ForwReach_{i+1}(x) \leftarrow ForwReach_{i+1}(x) \vee Succ_l(x)$

For abductive ineference the same (or an equivalent) minimal-cost plan can be obtained by chaining forward from the set of assumptions to the observations. An according implementation is shown in Algorithm 14.3.

The problem here is that the hypothesis Δ is not computed directly from the last set of states that has been reached, as it was inferred in backward search. However, after having extracted the step-minimal explanation, besides the set of actions, the set of states in the plan and especially the completion to A is also computed.

Having forward and backward inferences, it is also possible to operate bi-directional, reducing the complexity of finding the smallest explanation drastically. A coarse argumentation is that bi-directional breadth-first search with a goal distance d in a graph with uniform branching factor b (and no duplicate elimination) looks at $2b^{d/2}$ states, a number exponentially smaller than b^d, the efforts for unidirectional BFS.

14.8.3 Cost-Optimal Abductive Inference

In many cases, the principle of Occam's razor to compute the step-minimal explanation in abductive reasoning is insufficient. In other words, the relevance of operators for the inference process is not uniform. We assign costs $cost(o)$, $o \in O$, denoting how important individual actions are (the higher the cost, the less important the operator).

```
(:action rule-1 :parameters (?X ?Y - person)
 :precondition (and (like ?X ?Y) (ill ?Y) (irreplacable ?Y))
 :effect (and (depressed ?X) (increase (total-cost) 3)))
(:action rule-2 :parameters (?X - person)
 :precondition (and (pessimist ?X))
 :effect (and (depressed ?X) (increase (total-cost) 10)))
(:action rule-3 :parameters (?X - person ?O - organ)
 :precondition (and (has ?X ?O) (illness ?O))
 :effect (and (ill ?X) (increase (total-cost) 1)))
(:action rule-4 :parameters (?O - organ)
 :precondition (and (heart-attack ?O))
 :effect (and (illness ?O) (increase (total-cost) 3))))
```

Figure 14.4: PDDL Actions for Example Domain.

As an example we once again take the case of John being depressed. Fragments of the PDDL model are shown in Figure 14.4. If we assign costs to inference operators such as $cost(rule_1) = 3$, $cost(rule_2) = 10$, $cost(rule_3) = 1$, and $cost(rule_4) = 3$ we get the cost-minimal explanation that John is depressed because of Mary's heart attack.

Algorithm 14.4: Algorithm for cost-based backward abduction.

Procedure Backward Cost-based BDD-Abduction
Input: Cost-based planning problem with theory
$T = \bigvee_{o \in O} T_o$, set of assumptions $A \subseteq AP$,
and set of observations G
Output: Cost-optimal explanation Δ that is
consistent with $A \wedge \Delta \models_T G$

$BackReach_0(x) \leftarrow G(x)$
for each $i = 0, 1, \ldots$
  $Min(x) \leftarrow BackReach_i(x)$
  if $(Min(x) \wedge A(x) \neq false)$
    return $ConstructExplanation(Min(x) \wedge A(x))$
  for all $l = 1 \ldots, C$
    $Pred_l(x') \leftarrow$
      $\bigvee_{o \in O, cost(o)=l} (\exists x' (Min(x') \wedge T_o(x, x'))[x \leftrightarrow x']$
    $BackReach_{i+l}(x) \leftarrow BackReach_{i+l}(x) \vee Pred_l(x)$

A pseudo-code implementation is given in Algorithm 14.4. The core difference to uniform cost abduction is that the preimages of the transition relation are computed for each cost value l in $1, \ldots, C$. Zero-cost actions can be included by computing a transitive closure wrt all such actions before expanding a bucket.

Forward and backward induction simulate Dijkstra's algorithm. Bidirectional search now faces the problem that the first intersection of the search frontiers does not necessarily yield the minimum-cost solution. For such a case, symbolic perimeter search is applied as follows. In a first phase we construct a perimeter database, storing the backward layers up to some depth. This database then serves as a heuristic for guiding the search in forward direction.

14.8.4 Manual Selection Strategies

Having computed at least one valid hypothesis, it is conceptually easy to generate the corresponding plan. The simplest solution is to chain the sequence $BackReach_i$ down to $BackReach_0$ backwards starting with a state in

Algorithm 14.5: Bidirectional abduction with perimeter database.

Procedure Bidirectional Cost-based BDD-Abduction
Input: Cost-based planning problem with theory
$T = \bigvee_{o \in O} T_o$, set of assumptions $A \subseteq AP$,
and set of observations G, backward layers $BackReach_i$
Output: Cost-optimal explanation Δ that is
consistent with $A \wedge \Delta \models_T G$

for all $h \in \{0, \ldots, \max_h\}$
  $Open[0,h](x) \leftarrow A(x) \wedge BackReach_h(x)$
for all $f \in \{0,1,2,\ldots\}$, $g \in \{0,\ldots,f\}$
  $Min(x) \leftarrow Open[g, f-g](x)$
  if $(Min(x) \wedge G(x) \neq false)$
    return $ConstructExplanation(Min(x) \wedge\ G(x))$
  for all $i \in \{1,\ldots,C\}$
    $Succ_i(x') \leftarrow$
      $\bigvee_{o \in O, cost(o)=i} \exists x.\ Min(x) \wedge Trans_o(x,x')[x \leftrightarrow x']$
    for all $h \in \{0,\ldots,\max_h\}$
      $Open[g+i,h](x) \leftarrow Open[g+i,h](x) \vee$
        $Succ_i(x) \wedge BackReach_h(x)$

the intersection and computing forward images of one selected state in $BackReach_j$ that is intersected with the next state set $BackReach_{j+1}$.

This explanation can be returned to the user who refines the result by either strengthening or weakening the assumption A. Alternatively, he can simply reject the plan, giving rise to a Taboo list D that is subtracted from the goal, i.e. setting G to $G \setminus D$. Last but not least, we may allow him to eliminate the impact of certain operators from the plan by rescaling their influence.

14.8.5 Finding Critical Query Variables

For an interactive fault diagnosis with a small number of queries, it is important to reduce the uncertainty in the domain of the variables. One promising aspect is to query the variable that reduces the set of possible worlds by the largest margin, by means that the set of satisfying assignments to the variables is minimal.

For the problem of finding the most critical variable in *ValidHypothesis* we exploit the fact that model counting (the process of determining the number of satisfying assignments to a Boolean formula) in a BDD is efficient. Hence for each finite-domain variable v and each possible assignment i in the domain $Domain(v)$ of v we determine $\sum_{i \in Domain(v)} ModelCount(ValidHypothesis \wedge (v = i))$ and take the variable for which this quantity is the smallest.

14.9 Plan Recognition Experiments

For executing abductive reasoning we adapted the planner Gamer.

Grounding the planning domains yields a finite-domain variable encoding of the problems. The original results of fully specified benchmark problems under closed world assumption (cwa), meaning that the parts not mentioned in the intial state are false, are compared to an open world (no cwa), meaning that the parts not mentioned

Table 14.4: Results in Elevator.

	cwa					no cwa					half init				
	cost	steps	back.	forw.	total	cost	steps	backw.	forw.	total	cost	steps	backw.	forw.	total
1	42	14	2:31	0:06	2:38	42	14	2:30	0:08	2:40	15	6	2:29	0:07	2:38
2	26	9	2:30	0:05	2:37	26	9	2:29	0:05	2:37	19	6	2:29	0:05	2:37
3	55	18	2:30	0:06	2:38	55	18	2:30	0:06	2:37	27	10	2:29	0:05	2:37
4	40	18	2:29	0:09	2:40	40	18	2:29	0:09	2:41	20	9	2:29	0:08	2:40
5	55	22	2:30	0:09	2:40	55	22	2:29	0:09	2:40	43	16	2:29	0:13	2:45
6	53	26	2:29	0:57	3:28	53	26	2:29	0:57	3:29	25	14	2:29	0:07	2:38
7	62	27	2:29	0:42	3:14	62	27	2:29	0:37	3:08	27	12	2:30	0:04	2:36
8	53	25	2:28	2:14	4:46	53	25	2:29	2:16	4:48	30	12	2:29	0:33	3:05
11	56	17	2:29	0:05	2:36	56	17	2:30	0:05	2:36	34	10	2:30	0:05	2:36
12	54	16	2:29	0:06	2:38	54	16	2:30	0:06	2:38	37	11	2:29	0:09	2:41
13	59	17	2:30	0:07	2:38	59	17	2:30	0:07	2:38	40	10	2:30	0:06	2:37
14	63	22	2:29	0:21	2:53	63	22	2:30	0:20	2:52	39	11	2:29	0:12	2:44
15	66	24	2:30	0:12	2:44	66	24	2:30	0:13	2:44	40	14	2:30	0:08	2:40
16	–	–	–	–	–	–	–	–	–	–	43	15	2:28	2:16	4:48
17	–	–	–	–	–	–	–	–	–	–	42	13	2:28	0:14	2:47
21	48	17	2:30	0:07	2:39	48	17	2:30	0:07	2:39	31	10	2:30	0:06	2:38
22	54	19	2:29	0:37	3:09	54	19	2:29	0:38	3:10	36	12	2:30	0:46	3:17
23	–	–	–	–	–	–	–	–	–	–	39	15	2:29	0:18	2:49
24	56	24	2:29	2:10	4:42	56	24	2:29	2:12	4:44	–	–	–	–	–
25	63	27	2:29	0:57	3:29	63	27	2:29	1:00	3:32	43	20	2:30	2:28	5:00
26	–	–	–	–	–	–	–	–	–	–	29	12	2:29	1:33	4:06

in the intial state are unknown. Then we omit every second fact in the initial state (half init), i.e., we eliminate parts of the initial state (in an open world) in order to reconstruct it.

The abduction algorithm we chose was bidirectional cost-based BDD abstraction (Algorithm 14.5). The total time bound was set to five minutes, from which we took at most 150s for backward perimeter construction and the remaining time for forward search. In some domains (*ParcPrinter*, *PegSolitaire*, and *Sokoban*) dropping the closed world assumption leads to empty plans (the intersection of the initial state with the goal states is not empty), such that we dropped these examples from the presentation.

Table 14.4 shows the results in the *Elevator* domain. Here, we see that dropping the closed world assumption has no effect. This is an immediate consequence of the fact that the initial state is fully specified with positive literals. For a half-way specified initial state we see some advances. In some cases new problems could be solved, in others the abductive inference is harder.

Table 14.5 depicts the results in the *Openstacks* domain. Here we see that abduction gets harder. This is due to the fact that cost-based backward induction to construct the perimeter database does not provide any information, as cost zero results in no information. The forward abduction phase thus degrades to uniform cost search.

Table 14.6 shows the results in the *Transport* domain. Here we have considerably larger costs than in the previous ones. Table 14.7 gives the results in the *Woodworking* domain, showing a similar trend as the results in the *Transport* domain.

14.10 Summary

Heuristic and symbolic search are two leading methods for sequential optimal planning. In this chapter we have seen a comparison of two symbolic high-quality lower bounds for cost-optimal planning, namely the PDB and the M&S heuristic. Surprisingly, the former performance was superior (if we stick to a single planner run).

Table 14.5: Results in Openstacks (ADL).

	cwa					no cwa					half init				
	cost	steps	back.	forw.	total	cost	steps	backw.	forw.	total	cost	steps	backw.	forw.	total
1	2	17	0:01	0:01	0:02	1	11	0:01	0:01	0:02	0	6	0:00	0:00	0:02
2	2	20	0:01	0:01	0:02	1	13	0:01	0:01	0:02	0	7	0:00	0:00	0:02
3	2	23	0:01	0:01	0:03	1	15	0:01	0:01	0:02	0	8	0:00	0:00	0:02
4	3	27	0:01	0:01	0:04	1	17	0:01	0:01	0:04	0	9	0:01	0:00	0:03
5	4	31	0:02	0:01	0:04	1	19	0:02	0:01	0:04	0	10	0:01	0:01	0:04
6	2	32	0:04	0:01	0:08	1	21	0:05	0:01	0:07	0	11	0:04	0:01	0:07
7	5	38	0:13	0:02	0:16	1	23	0:13	0:01	0:16	0	12	0:12	0:02	0:15
8	5	41	0:33	0:02	0:37	1	25	0:35	0:02	0:38	0	13	0:34	0:02	0:38
9	3	42	0:47	0:02	0:51	1	27	0:51	0:02	0:54	0	14	0:51	0:04	0:56
10	3	45	2:30	0:03	2:34	1	29	2:30	0:03	2:34	0	15	2:29	0:19	2:51
11	4	49	2:30	0:05	2:36	1	31	2:30	0:03	2:35	0	16	2:29	0:48	3:20
12	3	51	2:30	0:08	2:40	1	33	2:30	0:04	2:36	0	17	2:29	0:54	3:26
13	4	55	2:29	0:26	2:59	1	35	2:29	0:07	2:40	–	–	–	–	–
14	4	58	2:29	0:11	2:44	1	37	2:29	0:12	2:44	–	–	–	–	–
15	–	–	–	–	–	1	39	2:29	0:13	2:45	–	–	–	–	–
16	4	64	2:29	1:20	3:52	1	41	2:29	1:12	3:44	–	–	–	–	–
17	–	–	–	–	–	1	43	2:29	2:00	4:33	–	–	–	–	–
18	3	69	2:29	0:43	3:16	–	–	–	–	–	–	–	–	–	–
19	4	73	2:29	2:21	4:54	–	–	–	–	–	–	–	–	–	–
22	4	82	2:28	0:29	3:02	–	–	–	–	–	–	–	–	–	–

Table 14.6: Results in Transport.

	cwa					no cwa					half init				
	cost	steps	back.	forw.	total	cost	steps	backw.	forw.	total	cost	steps	backw.	forw.	total
1	54	5	0:01	0:01	0:04	54	5	0:01	0:01	0:02	52	3	0:01	0:01	0:04
2	131	12	0:10	0:02	0:12	131	12	0:10	0:02	0:12	98	8	0:11	0:02	0:13
3	250	17	2:30	0:08	2:40	250	17	2:30	0:08	2:40	88	7	2:30	0:07	2:40
4	318	22	2:29	1:08	3:42	318	22	2:29	1:12	3:45	170	13	2:29	0:47	3:21
11	456	9	0:01	0:01	0:03	456	9	0:01	0:01	0:03	243	5	0:01	0:01	0:03
12	594	16	0:54	0:05	0:57	594	16	0:54	0:05	0:57	264	8	0:59	0:05	1:02
13	550	21	2:29	0:25	2:58	550	21	2:29	0:25	2:58	294	11	2:30	0:38	3:11
21	478	7	0:00	0:00	0:02	478	7	0:00	0:00	0:02	160	3	0:00	0:00	0:02
22	632	12	0:13	0:03	0:16	632	12	0:15	0:03	0:17	431	8	0:15	0:03	0:17
23	630	17	2:30	0:09	2:42	630	17	2:30	0:10	2:42	265	7	2:30	0:09	2:41
24	614	19	2:28	0:53	3:26	614	19	2:28	0:57	3:30	231	9	2:29	0:13	2:47

The variable ordering for the M&S heuristic influences both the quality of the estimate and the symbolic exploration. The heuristic choice applied in FD pleases the M&S heuristic, while the optimization applied in Gamer pleases symbolic exploration. One could combine the two for a competitive BDDA* exploration with the M&S heuristic. The small ADD sizes for the M&S heuristic suggest that there is sufficient memory for computing the maximum of more than one heuristic (in ADD representation). This results in a consistent, strictly more informed heuristic for the (BDD)A* exploration and provides a way of combining the accuracy of PDBs and M&S heuristics. In many instances that are solved by BDDA* with PDBs no abstraction is applied, meaning that blind symbolic backward search in the concrete state space is either finalized or truncated by the time limit. As a consequence at least in domains where backward search does not explode immediately (due to illegal states produced), bidirectional blind symbolic search is best.

Abductive reasoning selects hypotheses that explain the evidences best. It is of high relevance for AI, but – due to its large complexity demands – received limited attention in the last decade. As computational power on modern CPUs and planner technology have improved substantially, we showed a promising technique to compute valid hypotheses time- and space-efficiently. The focus is the hypothesis generation problem. Moreover, we discussed different options for interaction to overcome the limitations of Occam's razor, including interac-

Table 14.7: Results in Woodworking.

	cwa					no cwa					half init				
	cost	steps	back.	forw.	total	cost	steps	backw.	forw.	total	cost	steps	backw.	forw.	total
1	170	9	0:01	0:01	0:04	170	9	0:01	0:01	0:04	75	5	0:02	0:01	0:04
2	185	9	0:04	0:02	0:07	185	9	0:04	0:02	0:08	135	8	0:05	0:02	0:08
3	275	18	1:07	0:07	1:17	275	18	1:02	0:08	1:12	80	8	1:11	0:04	1:17
4	–	–	–	–	–	–	–	–	–	–	100	8	2:29	0:06	2:38
11	130	7	0:01	0:01	0:04	130	7	0:01	0:01	0:04	45	4	0:01	0:01	0:04
12	225	12	0:13	0:04	0:19	225	12	0:11	0:04	0:18	130	7	0:14	0:03	0:19
13	215	13	2:05	0:06	2:14	215	13	1:57	0:06	2:07	110	7	2:05	0:05	2:13
14	225	16	2:30	1:24	3:56	–	–	–	–	–	145	8	2:30	0:05	2:38
16	–	–	–	–	–	–	–	–	–	–	120	8	2:29	0:14	2:47
21	95	6	0:01	0:01	0:04	95	6	0:01	0:01	0:04	35	2	0:01	0:01	0:03
22	185	9	0:04	0:02	0:07	185	9	0:04	0:02	0:07	105	5	0:04	0:01	0:07
23	195	13	0:23	0:03	0:28	195	13	0:23	0:03	0:28	95	7	0:26	0:02	0:30
24	245	15	2:29	0:14	2:47	245	15	2:29	0:19	2:51	145	7	2:29	0:05	2:38

tion based modified operator costs, altering the set of assumptions, maintaining tabu lists for goals and applying model-counting. They all help to reduce the amount of uncertainty.

How much does such abductive planning differ from ordinary symbolic planning? Besides the initial state being partial, there is not much change in the exploration algorithms. This is true, but it also highlights the advantage symbolic has wrt explicit-state planning: the latter would have to enumerate all possible completions either for the initial state in order to perform the inference process. Moreover, most existing explicit-state planning systems are not cost-optimal. One difference between abduction and ATMS inference is that the latter logs justifications to assignments to allow multiple fault analyses. From a logical perspective, abduction chains backwards in time, from the set of observations towards the set of assumptions, while the inference in ATMS is multi-directional, depending on the update to the set of assignments to an incident variable of a device, while propagating the effects through the network.

14.11 Bibilographic Notes

Among the leading heuristics in AI planning are pattern databases, invented by [143] and applied to action planning by [178]. Implicit pattern databases were studied by [388] while partial pattern databases have been contributed by [17]. The automated selection of state-space abstractions has been considered by [178, 180, 324, 402]. Planning pattern databases (PDBs) map the state space of a classical planning task onto a smaller abstract state space by considering only a subset of the task's variables, which is called a pattern [143, 178]. The combination of several PDBs can result in better cost-to-go estimates than the estimates provided by each PDB alone. One has combined multiple PDBs by taking the maximum [344, 33] or the sum [253] of the PDBs' estimates. In this chapter, we also considered the canonical heuristic function, which takes the maximum estimate over all additive PDB subsets [324]. The challenge is to find a collection of patterns from which an effective heuristic is derived.

Multiple approaches have been suggested to select good pattern collections [324, 180, 402]. One work showed that using a genetic algorithm [180] to generate a large collection of PDBs and greedily selecting a subset of them can be effective in practice [444]. However, while generating a PDB heuristic, Lelis *et al.*'s approach is blind to the fact that other PDBs will be considered in the selection process. The proposed method, which we call Complementary PDBs Creation (CPC), adjusts its PDB generation process to account for the PDBs already generated as well as for other heuristics optionally provided as input.

Other state-of-the-art heursitics are *LM-cut* and *Merge&Shrink*. The Merge&Shrink planning heuristic has been introduced by [333]. Further proposals for this heuristic improve its quality [495, 390] and outperform other

state-of-the-art heuristics like *LM-cut* [332] in many benchmarks. Improvements based on bisimulation reductions can be found in [495]. Merge&Shrink was originally proposed in the context of directed model checking [167, 168]. Some competitors combined multiple planners or heuristics, others used new methods like flow heuristics, automated pruning via symmetries and partial-order reduction or incremental computation of *LM-cut* [125].

The combination of symbolic exploration and A* search was shown to be effective [217] and SetA* [367]. Partitioned ways of computing the successor set were applied by [619] and mutex constraints were used to simplify the problem and rule out illegal states by [616]. On top of these advances, the best planners had the following ingredients: Gamer variants used symbolic bidirectional blind search: Dynamic-Gamer uses dynamic variable reordering during the search [403] and cGamer extends Gamer with state-invariant constraints and improved successor generation methods; SymBA* is a combination of forward and backward symbolic heuristic searches and uses perimeter abstraction heuristics [162]. for the search frontiers to meet and to finally prove optimality; SPM&S applies A^* search with symbolic perimeter abstraction heuristics. It performs symbolic regression to construct a perimeter around the goal, which is used to initialize a symbolic version of PDBs and *Merge&Shrink* heuristics [620]; RIDA* is a non-symbolic planner that combines different heuristics, selecting a subset of useful heuristics for each given problem [33]. Among other heuristics, such as *LM-cut*, it uses genetic algorithms to generate a large set of PDBs [180].

BDDs go back to work by Akers and Bryant [89, 651], while ADDs were introduced by [31]. Theoretical results on BDD in planning have been shown by [206]. BDDs are especially good in the construction of PDBs [178] and related estimates [212]. They support breadth-first and cost-first search in both directions [402]. Algorithms like BDDA* [217] use BDDs for the search and for the heuristic. Symbolic PDB heuristics [179] refer to backward unguided exploration in abstract state spaces. Other heuristics like *Merge&Shrink* also enjoy a BDD representation [213]. Minato et al. [477] have illustrated how to store several BDDs in a joint structure. One of the most widely used BDD libraries CUDD is maintained by Fabio Somenzi. Many aspects to the theory of decision diagrams have been given by Wegener [650]. BDDs are less compact than other structures like d-DNNFs [147], but offer a unique representation of Boolean functions.

The planner Complementary is an updated and significantly modified version of [269].

Plan recognition problems have been addressed by means of different approaches. Dynamic Bayesian networks [12] have been applied in a multi-user dungeon game, and relational Markov models (RMM) [15] for the recognition of user actions in adaptive web interfaces. Although these approaches have been successful even in the face of noisy observations, their use is limited to specific areas of application. While Bayesian models support the representation of complex structural dependencies they lack the ability to represent relation properties in terms of a RMM and vice versa. Otherwise, more expressive probabilistic representations naturally result in a significant decrease in efficiency. In contrast, classification-based approaches (e.g., [361]) allow more expressive/flexible representations, but are limited to domains with precisely given sets of possible plans/intentions like, e.g., in American football.

Peirce [507] defines abduction as the process of finding the cause for a set of assumptions and a theory provided. Its philosophical roots go back to Aristotle, while for AI [518] abduction is commonly viewed as a form of reasoning, allowing one to find explanations [493].

For cost-based abdution using BDDs [386], in the logical context of propositional Horn clauses, the authors compile a BDD for the theory. Each BDD variable corresponds to a different hypothesis. BDD edges are associated with costs, and additional consistency criteria lead to pruning of edges in the BDD. Satisfying paths in the BDDs correspond to the set of possible hypotheses (0-Edges are neglected). As the BDD for the entire theory may be too large, alternative data structures are proposed that handle inference rules lazily. (Cost-annotated) Horn inferences are much more restricted than (cost-annotated) SAS^+ planning inferences that are considered here. While the former can be solved by variable substitution in the BDD, the latter requires exploration with BDDs.

Especially when applied to business settings, checking for anomalies in a knowledge base becomes a very important task. The efficiency of labeling clearly depends on the compactness of the generated labels. As these may require exponential size, with the exponent being in the depth of the rule sets, more efficient representations like

BDDs are needed [615, 486]. Such symbolic approaches encode the system's input in binary form and traverse the rule base, thereby constructing the BDDs instead of labels that describe the in- and output dependencies of the system, checking BDDs against each other and reporting any observed anomaly. Alternative compilations of knowledge bases are possible [147].

In diagnosis, we are not only concerned with detecting errors, but additionally with explaining them. For multiple faults, assumption-based truth maintenance systems (ATMSs) have been suggested [265]. The model is an undirected network with the edges labeled with discrete variables, whose values are of a certain range. Devices in the network to be diagnosed propagate the information found at incident edges.

The importance of BDDs for covering the amount of uncertainty through compactly representing all possible worlds is immediate [46]. Concise encodings exploit that usually sets of possible assignments to variables are restricted to small ranges. Probing an edge is an assignment to a variable that reflects the background knowledge received in an interaction with the user.

Abduction has taken on fundamental importance in AI [483] including planning [29], database updates [379], text understanding [496], and others. Some diagnosis problems [265] may be solved with abduction, given that the set of possible misbehaviors is encoded in the model.

One fundamental problem of abduction is its complexity. For logic-based abduction, Eiter [238] classified many problems to lie in the second or third level of the polynomial hierarchy, while [95] proved the NP-hardness for pure propositional problems to explain a set of data best. There is a balance between the flexibility in the modeling language and the inferences that can be drawn. The problem of generating abductive explanations is usually divided into two subproblems that can be addressed separately: 1) generating the set of all possible explanations, and 2) selecting the most appropriate hypothesis among the set of possible explanations. Efficient approaches to abduction are limited. A tractable solution to the generation problem is limited to Horn theories and positive observation literals.

The second problem has been addressed in very different ways. The most widely used selection criteria is Occam's razor [612]. It states that for two explanations, the simpler one is preferable. A different domain-independent criterion has been applied [493] in the domain of text understanding, *the coherence metric*. In addition different *domain-specific* solutions have been proposed [20, 339].

The logic-based inference is first-order; it is computationally intractable [238]. Even for the propositional case many problems like minimization are hard [95]. Abductive inference is not necessarily limited to logical representations. For an overview see [505]. An inherent weakness of the logic-based approach to abduction is the very specific interpretation of the logical implications. Abductive inference assumes that logical implication encodes causal knowledge (relations). Although this property may hold in some scenarios (e.g., in diagnostic domains) it is clearly not valid in general. As a consequence, abductive inference often leads to non-causal explanations. We claim to overcome this problem. Instead of Horn or first-order logic programs as the basis for backward inference, we chose PDDL as the modeling language, which is automatically converted to a plan model with discrete state variables. Thereby, we obtain access to a wealth of planning benchmarks to be exploited for abductive reasoning. In contrast to many other approaches to logical abduction we do not address the problem of hypothesis generation and hypothesis selection independently. Instead of calculating the set of all possible explanations we directly calculate the *best* hypothesis. We apply Occam's razor in form of cost-optimal explanation as the fundamental selection criterion. The presented approach extends to domains-specific criteria in terms of *weighted* abduction [20, 339].

Chapter 15
General Game Playing

In recent years, general game playing has received an increasing amount of attention. In general game playing the agents are provided a description of a game according to certain rules and need to play it. General game playing also allows us to express multi-player games and supports any number of participants. In case of multi-player games, the agents often play against each other, while in case of single-player games one agent tries to find a way to reach a terminal state where it can achieve the best reward possible. The authors of the agents do not know which games will be played, so no domain-specific knowledge can be inserted. In general game playing the players only get rewards for achieving goals: for each possible terminal state the player is awarded points ranging from 0 (worst) to 100 (best).

Our first goal is to strongly solve the games, i.e., we want to find the outcome for each player in any reachable state in case of optimal play. Using domain-dependent solvers, this has often been done in the past. One of the most prominent results was that the outcome of American Checkers is a draw. Of course, without domain-specific knowledge, we cannot expect to come up with solutions for such complex games in general game playing.

We also present a full-fledged player for general games with incomplete information. To deal with uncertainty we introduce a method that operates on sets of belief states. For searching for a set of belief states we present depth-first and Monte-Carlo methods. All can be combined with any traditional general game player, e.g., using minimax or UCT search.

15.1 Introduction

General[1] game playing (GGP) urges the computer to process the rules of the game and start to play, thus operating without including expert knowledge of the game that is played. In the context of International GGP Competitions the rules are specified in a logical formalism, the *game description language* (GDL). The games are played on a server, which connects with game playing agents via TCP/IP. After some startup time the game starts and according to a playclock moves have to be issued. Moves are executed on the server and reported to the players to continue. An example GDL domain description for TicTacToe is given in Program 15.1.

As randomness and handling incomplete information are necessities for playing many games (e.g., card games), GDL has been extended to GDL-II (short for GDL with incomplete information). Even though benchmark games can be played, some of their complexities, especially due to the additional efforts required to handle incomplete information, do not match well with the ones that players can handle well. Players act in the so-called *belief state space* and some assumptions of the current state might become invalid due to incoming information provided by the server.

[1] This chapter is based on joint work with Peter Kissmann and Tim Federholzner. It puts together and improves the work from [401, 401, 192].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_15

Program 15.1: GDL specification of the game TicTacToe.

```
(role xplayer) (role oplayer) ; names of the players

(init (cell 1 1 b)) ... (init (cell 3 3 b)) ; all cells empty
(init (control xplayer)) ; xplayer is active

(<= (next (cell ?m ?n x)) (<= (next (cell ?m ?n o)) ; effect of marking a cell
    (does xplayer (mark ?m ?n))) (does oplayer (mark ?m ?n)))
(<= (next (cell ?m ?n ?w)) ; part of the frame (marked cells remain marked)
    (true (cell ?m ?n ?w)) (distinct ?w b))
(<= (next (cell ?m ?n b)) ; part of the frame (untouched empty cells remain empty)
    (does ?w (mark ?j ?k)) (true (cell ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))
(<= (next (control xplayer)) (<= (next (control oplayer)); change of the active player
    (true (control oplayer))) (true (control xplayer)))

(<= (legal ?w (mark ?x ?y)) ; possible move (empty cell can be marked)
     (true (cell ?x ?y b)) (true (control ?w)))
(<= (legal xplayer noop) (<= (legal oplayer noop) ; if opponent active, no move
    (true (control oplayer))) (true (control xplayer)))

; axioms for reducing complexity of description
(<= (row ?m ?x)
    (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
(<= (column ?n ?x)
    (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
(<= (diagonal ?x)
    (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
(<= (diagonal ?x)
(true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
(<= (line ?x) (row ?m ?x)) (<= (line ?x) (column ?m ?x)) (<= (line ?x) (diagonal ?x))
(<= (goal xplayer 100) (line x)) ; rewards for xplayer (oplayer analogously )
(<= (goal xplayer 50) (not (line x)) (not (line o)))
(<= (goal xplayer 0) (line o))

; terminal states
(<= terminal (line x)) (<= terminal (line o)) (<= terminal (not (true(cell ?m ?n b))))
```

15.2 General Game Playing

15.2.1 GDL

As the full description of syntax and semantics in GDL is involved, we prefer the following set-based definition of a general game. A general game is a tuple of the form $(P, S, s_0 \in S, T \subseteq S, M, succ, reward)$, where

- P is the set of roles (the players),
- S is the set of all states,
- s_0 is the (unique) initial state,
- T is the set of terminal states,

- M is the set of moves,
- *succ* is a mapping $M^{|P|} \times S \to S$ for the calculation of the successor states, and
- *reward* is a mapping $P \times T \to \{0, \ldots, 100\}$, that defines the rewards of the player at the end of the game.

The difficulty of the Minimax-based approaches is to find a good evaluation function that will work well for any game, as most games cannot be fully analyzed in the available time.

When a leaf node is reached in a UCT player it is expanded and a simple Monte-Carlo run is started, which chooses the moves uniformly at random until a terminal state is reached. Afterward, the reward achieved at the reached terminal state is propagated through the tree and the average rewards of all states expanded in the last simulation run are updated accordingly.

15.2.2 GDL-II

Syntactically, GDL-II is a small extension. It mainly adds two predicates. The first one is an additional role in P, called *random*. The other one is the predicate *sees* that defines the visibility of information to the players. While before the players were informed of the moves that all participating players have chosen, in case of GDL-II the players are informed only about what they can see. Nevertheless, the players should always be able to determine the set of legal moves they currently may choose and when a terminal state has been reached.

Let us consider some fragments of a simple coin flipping example. At any time, the random player can only flip two coins (see Program 15.2). The two possibilities are *heads* and *tails*. As it is the random player, we know that it will choose each with the same probability.

Program 15.2: Legal moves of random in the coin flipping example.

```
(role random)
(<= (legal random (flip ?coin1 ?coin2))
   (coin ?coin1) (coin ?coin2)
)
(coin heads) (coin tails)
```

If the coin shows *heads* and the player has chosen *go*, it is moving forward (cf. Program 15.3). However, if the coin shows *tails*, the player is blocked and thus does not change the position. If it chose to *stay*, it also will stay at the previous position.

Program 15.3: Update of the positions of the players in the coin flipping example.

```
(<= (next (position white ?n2))
   (does white go) (does random (flip heads ?other))
   (true (position white ?n1)) (next_pos ?n1 ?n2)
)
(<= (next (position black ?n2))
   (does black go) (does random (flip ?other heads))
   (true (position black ?n1)) (next_pos ?n1 ?n2)
)
(<= (next (position white ?n))
   (does white go) (does random (flip tails ?other))
   (true (position white ?n))
)
(<= (next (position black ?n))
   (does black go) (does random (flip ?other tails))
   (true (position black ?n))
)
(<= (next (position ?r ?n))
   (does ?r stay)
   (true (position ?r ?n))
)
```

To determine if the own player (and the opponent) was able to move the observations are sent according to the *sees* rule (cf. Program 15.4). This informs only about the results of the coin flip performed by the random player, so that the actual position of the own player must be evaluated based on the performed moves, while that of the opponent is unknown, as the player does not know if it chose to go or stay.

Program 15.4: The information the players see in the coin flipping example.

```
(<= (sees ?r (did random (flip ?coin1 ?coin2)))
   (role ?r)
   (does random (flip ?coin1 ?coin2))
)
```

As GDL-II search control heuristics there is a much wider spectrum of possibilities. One can maximize the own flexibility, i.e., to prefer nodes with large branching factor, in order not to get stuck too quickly, or to minimize the branching factor of the opponent. Such heuristics are to be implemented with care. While flexibility is often good for games like Chess, in Checkers pieces are sacrificed too quickly. A better rule of thumb for GDL-II games is to maximize the own knowledge and minimize the one offered to the opponents. But the former can also lead to too many sacrifices like in Kriegspiel and the latter to avoid conflict that must be resolved. As with the no-free lunch theorems in optimization there is hardly a heuristic that is effective for all games.

For symbolic search, we need BDDs to represent the initial state $\mathscr{I}$, the terminal states $\mathscr{T}$, the formula describing when the players get which reward $\mathscr{R}$, as well as the moves $\mathscr{M}$. Unfortunately, most games contain variables, so that we do not know the exact size of a state, but this information is mandatory for BDDs. If we perform some instantiation, we come up with a variable-free format. As all formulas are Boolean, generating BDDs of these is straight-forward. Figure 15.1 shows BDDs for some of the utility functions needed to evaluate the termination of Tic-Tac-Toe.

To decrease the number of BDD variables, we try to find groups of mutually exclusive predicates. For this we perform a simulation-based approach that identifies the input and output parameters of each predicate. Often,

input parameters denote the positions on a game board while the output parameters specify its content. Predicates sharing the same name and the same input but different output parameters can never be true at the same time and thus are mutually exclusive. If we find a group of n mutually exclusive predicates, we need only $\lceil \log n \rceil$ BDD variables to encode these.

After instantiation, we know the precise number of moves of all the players and can also generate the possible combinations of moves of all players, which results in $\mathscr{M}$. Each move $m \in \mathscr{M}$ can be represented by a BDD $trans_m$, so that the complete transition relation *trans* is the disjunction of all these: $trans := \bigvee_{m \in \mathscr{M}} trans_m$.

To perform symbolic search, we need two sets of variables: one set, S, for the current states, the other one, S', for the successor states. To calculate the successors of a state set *from*, in symbolic search we use the *image* operator:

$$image\,(from) := \exists S.\,\left(trans\,\left(S,S'\right) \wedge from\,(S)\right).$$

As these successors are represented using only S', we need to swap them back to S.[2] This way, if we start at the initial state, each call of the image gives us an entire BFS layer. So, BFS is simply the iteration of the image, until a fix-point is reached.

As the transition relation *trans* is the disjunction of a number of moves, it is equivalent to generate the successors using one move after the other and afterwards calculate the disjunction of all these states:

$$image\,(from) := \bigvee_{m \in \mathscr{M}} \exists S.\,\left(trans_m\,\left(S,S'\right) \wedge from\,(S)\right).$$

This way, we do not need to calculate a monolithic transition relation, which takes time and often results in a BDD too large to fit into RAM.

The inverse operation of the image is also possible. The *pre-image* results in a BDD representing all the states that are predecessors of the given set of states *from*:

$$pre\text{-}image\,(from) := \exists S'.\,\left(trans\,\left(S,S'\right) \wedge from\,\left(S'\right)\right).$$

With this, we can perform a BFS in backward direction as well.

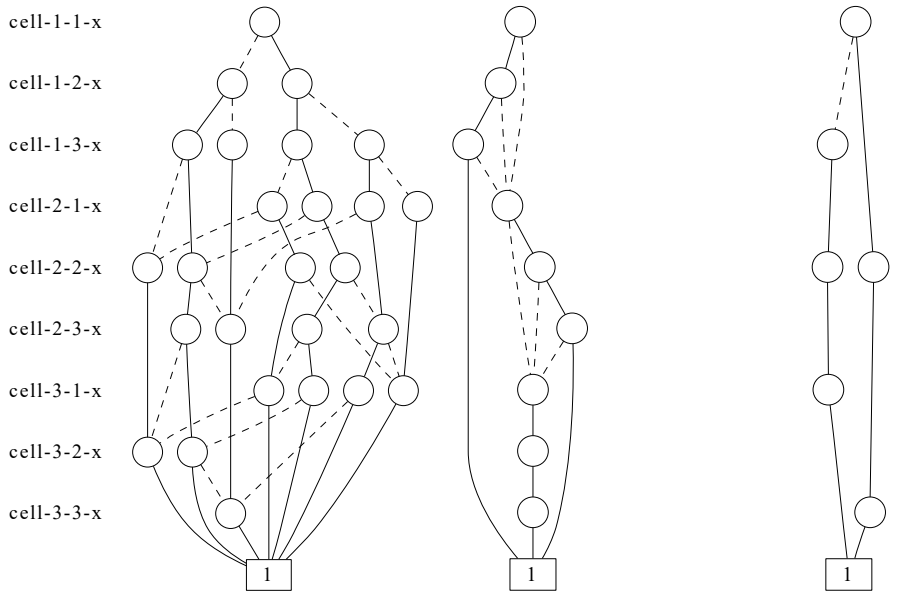
15.3 Solving General Two-Player Turn-Taking Games

In this section we show an algorithm to solve general two-player turn-taking games symbolically with only images and pre-images.

An existing approach works by using a 101×101 matrix M of BDDs. The BDD at $M[i,j]$ represents the states where player 1 can achieve a reward of i and player 2 a reward of j, $i,j \in \{0,\ldots,100\}$. Initially, all terminal states are inserted in the corresponding buckets. Starting at these, the strong pre-image is used to calculate those preceding states whose successors are all within the matrix and thus already solved. These predecessors are then sorted into the matrix by using the pre-image from each of the buckets in a certain order.

The refined algorithm works in two stages. First, we perform a symbolic BFS in forward direction (see Algorithm 15.1). Starting at the initial state, we calculate the successors of the current BFS layer by using the image operator. In contrast to the existing approach where a BFS was used to calculate the set of reachable states, here we retain only the BFS layers to partition the BDDs according to the layers the states reside in, hoping that the BDDs will stay smaller.

[2] We omit the explicit mention of this in the pseudo-codes to enhance readability. Whenever we write of an image (or pre-image), we assume such a swapping to be performed immediately after the image (or pre-image) itself.



(a) Column for xplayer. (b) Row for xplayer. (c) Diagonal for xplayer.

Figure 15.1: BDDs for the three of the utility functions of Tic-Tac-Toe used in the terminal states. Each node corresponds to a predicate (denoted on the left); solid edges mean that the predicate is true, dashed edges mean it is false. The bottom-most node represents the 1-sink, i. e., all paths leading from the top-most node to this sink represent satisfied assignments. The 0-sink has been omitted for better readability.

Algorithm 15.1: Calculate Reachable States (*reach*).

Input: General game description $\mathscr{G}$.
Output: Maximal reached BFS-layer.
$curr \leftarrow \mathscr{I}$
$l \leftarrow 0$
while $curr \neq \bot$ **do**
  store *curr* as layer l on disk
  $prev \leftarrow curr \wedge \neg \mathscr{T}$
  $curr \leftarrow image(prev)$
  $l \leftarrow l + 1$
return l

For the game TicTacToe, we start with the empty board. After one iteration through the loop, *curr* contains all states with one `x` being placed on the board; after the next iteration all states with one `x` and one `o` being placed, and so on.

Unfortunately, for the second step to work correctly, we need to omit duplicate detection (except for the one that implicitly comes with using BDDs). The search will terminate nonetheless, as the games in general game playing are finite but it might be possible to expand states more than once, if they appear on different paths in different layers.

A question that immediately arises is, when will we have to deal with such duplicate states. To answer this, we define a progress measure.

Let $\mathscr{G}$ be a general two-player turn-taking game and ψ be a mapping from states to numbers, i. e., $\psi : \mathscr{S} \mapsto \mathbb{N}$.

If $\mathscr{G}$ is not necessarily alternating, ψ is a *progress measure* if $\psi(s) < \psi(s')$ for all $(s,s') \in \mathscr{M}$. It is an *incremental progress measure*, if $\psi(s) = \psi(s') - 1$.

Algorithm 15.2: Solving general two-player games.

Input: General game description $\mathscr{G}$.
$l \leftarrow reach(\mathscr{G})$
while $l \geq 0$ **do**
  $curr \leftarrow$ load BFS layer l from disk
  $currTerms \leftarrow curr \wedge \mathscr{T}$
  $curr \leftarrow curr \wedge \neg currTerms$
  foreach $i, j \in \{0, \ldots, 100\}$ **do**
    $terms_{l,i,j} \leftarrow currTerms \wedge \mathscr{R}_{i,j}$
    store $terms_{l,i,j}$ on disk
    $currTerms \leftarrow currTerms \wedge \neg terms_{l,i,j}$
  for each $i, j \in \{0, \ldots, 100\}$ **do in specific order**
    $succ_1 \leftarrow$ load $terms_{l+1,i,j}$ from disk
    $succ_2 \leftarrow$ load $rewards_{l+1,i,j}$ from disk
    $succ \leftarrow succ_1 \vee succ_2$
    $rewards_{l,i,j} \leftarrow curr \wedge pre\text{-}image(succ)$
    store $rewards_{l,i,j}$ on disk
    $curr \leftarrow curr \wedge \neg rewards_{l,i,j}$
  $l \leftarrow l - 1$

Otherwise, ψ also is a *progress measure*, if $\psi(s) = \psi(s') < \psi(s'')$ for all $(s,s') \in \mathscr{M}_1$ and $(s',s'') \in \mathscr{M}_2$. It is an *incremental progress measure*, if $\psi(s) = \psi(s') = \psi(s'') - 1$.

Next we see, that whenever there is an incremental progress measure ψ for a general game $\mathscr{G}$, no duplicate arises within the different layers found by Algorithm 15.1.

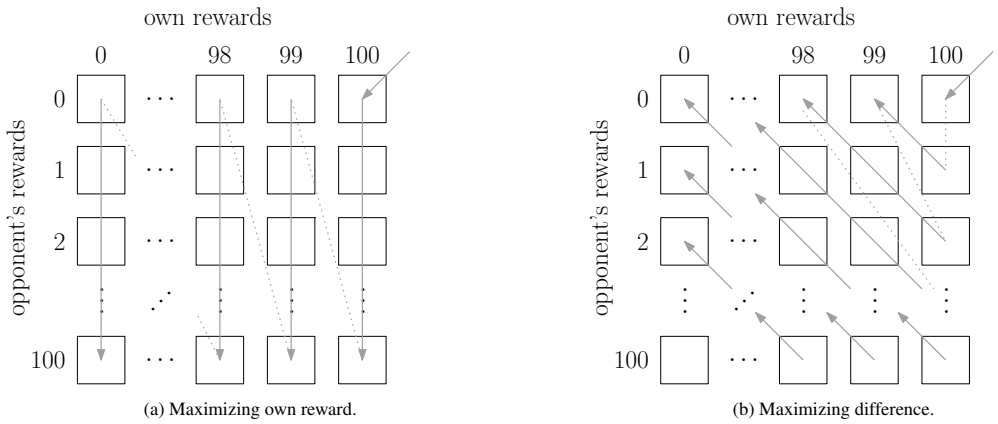
We need to show this for the two cases:

If $\mathscr{G}$ is not necessarily alternating, all states within one layer have the same progress measurement but a different one from any state within another layer. This can be shown by induction: The first layer consists only of the initial state. Let $succ(s)$ be the set of successor states of s, i. e., $succ(s) = \{s' | (s,s') \in \mathscr{M}\}$. According to the induction hypothesis, all states in layer l have the same progress measurement. For all states s in layer l and successors $s' \in succ(s)$, $\psi(s') = \psi(s) + 1$. All successors $s' \in succ(s)$ are inserted into layer $l+1$, so that all states within layer $l+1$ have the same progress measurement. It is also greater than that of any of the states in previous layers, as it always increases between layers, so that it differs from the progress measurement of any state within another layer.

Otherwise, the states within any succeeding layers differ, as the predicate denoting the active player has changed. Thus, it remains to show that for all $s, s' \in \mathscr{S}$, $s_1 \in \mathscr{S}_1$ and $s_2 \in \mathscr{S}_2$, $\psi(s) = \psi(s')$ if s and s' reside in the same layer and $\psi(s_1) = \psi(s_2)$ if s_1 resides in layer l and s_2 resides in layer $l+1$. For all other cases, the progress measurement of any two states does not match. The proof is very similar: The first layer consists only of the initial state. All successors of this state reside in the next layer and their progress measure equals, according to the definition of ψ. Let l be a layer that contains only states from $\mathscr{S}_1$. According to the induction hypothesis, all states in this layer have the same progress measurement. For all states s in layer l and successors $s' \in succ(s)$, $\psi(s) = \psi(s')$. All successors s' are inserted into layer $l+1$. For all states s' in layer $l+1$ and $s'' \in succ(s')$, $\psi(s'') = \psi(s') + 1$. All successors $s'' \in succ(s')$ are inserted in layer $l+2$, so that all states within layer $l+2$ have the same progress measurement. It is also greater than that of any of the states in previous layers, as it never decreases, so that it differs from the progress measurement of any state within different layers.

Note that in games that do not incorporate an incremental progress measure we need to expand each state at most d_{max} times, with d_{max} being the maximal distance from the initial state to one of the terminal states. This is due the fact that in such a case each state might reside in every layer.

Once all BFS layers are calculated we can start the second stage, the actual solving process, for which we perform a symbolic retrograde analysis (see Algorithm 15.2). We start at the last generated BFS layer l and move upwards layer by layer until we reach the initial state ($l = 0$).



(a) Maximizing own reward.

(b) Maximizing difference.

Figure 15.2: Order to traverse the reward combinations.

For each layer we perform two solving steps. First, we calculate all the terminal states that are contained in this layer. For these we then determine the rewards that the players get and store them in the corresponding files. As each player achieves exactly one reward for each possible terminal state, no specific order is needed in this step.

In the second step, we solve the non-terminal states. For this, we need to proceed through all possible reward combinations in a specific order. This order corresponds to an opponent model. The two most reasonable assumptions are that an agent either wants to maximize its own reward or to maximize the difference to the opponent's reward. The order, in which these reward combinations are processed, is indicated in Figure 15.2. For the experiments we assumed both players to be interested in maximizing the difference in the opponents' reward.

We load the BDDs representing the states that are terminal states or solved non-terminal states in the successor layer for which the players can surely achieve the corresponding rewards. From the disjunction of these we calculate their predecessors (using the pre-image). These states achieve the same rewards (in case of optimal play according to the opponent model) and thus can be stored on disk and must be removed from the unsolved states to prevent them from being assigned other rewards as well.

For the game Tic-Tac-Toe we start in layer 9, where all cells are filled. All these states are terminal states, thus we can solve them immediately by checking the rewards. Thus, we partition this layer into three parts: Those states where `xplayer` gets 100 points and `oplayer` 0 (a line of `x`s was established), those with the inverse score (a line of `o`s was established), and those with 50 points for each player (no line for any player).

In the next iteration, we reach those states where four `x`s and four `o`s reside on the board and the `xplayer` had control. First we remove those states containing a line for one of the players, as these are the terminal states, and solve them according to their rewards. Next, we check how to solve the remaining states. Thereto, we load the terminal states from layer 9 where the `xplayer` achieved 100 points, calculate their predecessors and verify if any of these predecessors is present in the set of the remaining states. If that is the case, we can remove them and store them in a file that specifies that the `xplayer` achieves 100 points and the `oplayer` 0 points for these states as well. In the Tic-Tac-Toe example, these are all the states where the placement of another `x` finishes a line. All the remaining states are draw states (as the `xplayer` can only place an `x` on the board and thus never finish a line of `o`s).

The presented algorithm is correct, i.e., it determines the game theoretical value wrt the chosen opponent model. The correctness of the forward search comes immediately from the use of a BFS. We generate all reachable states, no matter if we remove duplicates or not.

For the second stage, we need to show that all states are correctly solved according to the opponent model. We show this using induction. We start at the states in the final layer, which we immediately can solve according

Table 15.1: Results of solving two-player turn-taking games.

Game	Optimal Outcome
Catch a Mouse	100/0
Chomp	100/0
Clobber 3×4	0/40
Clobber 4×5	30/0
Connect 4 (5×6)	50/50
Cubi Cup 5	100/0
Nim	100/0
Number Tic-Tac-Toe	100/0
Sheep and Wolf	0/100
Sum 15	50/50
Tic-Tac-Toe	50/50

to their corresponding rewards. When tracing back towards the initial state, the terminal states again are immediately solvable by their rewards. The most important observation is that due to the construction, non-terminal states have successors only within the next layer. All states within this layer are already solved. If we check if a state has a successor achieving a certain reward and look at the rewards in the order according to the opponent model, we can be certain that all states within the current layer can be solved correctly as well.

Note that if we removed the duplicate states within different layers, we would reach states whose successors are not in the next layer but in some layer closer to the initial state and thus not solved yet, so we could not correctly solve such a state when reaching it for the first time.

Some games are not strictly alternating, i. e., a player might perform two or more consecutive moves, so that both players can be active in different states within the same BFS layer. To handle this, we split the second step of Algorithm 15.2 in two and perform this step once for the states where the first player was the active one and once for the second player. Note that both players go through the possible reward combinations in different orders, thus it is not possible to combine these two steps. Instead, we have to solve the states once for one player, store the result on disk, solve the remaining states for the other player, load the previous results, calculate the disjunction, and store the total result on disk. The order in which the two players are handled is irrelevant, as there is no state where both players are active.

For Clobber we specified rewards dependent on the number of pieces left on the board, so that we came up with general rewards, while the other games are all zero-sum games.

The runtime results for the new approach as well as the existing one are compared in Table 15.1. From this we can see that for the small games such as Tic-Tac-Toe or Sum 15 the new approach does not lose much, although all results are stored on the hard disk. Omitting this in the cases where all BDDs easily fit into RAM, however, would speed up the search.

For two slightly larger games, namely Chomp and Nim, the new approach is slower than the old one. This is due to the fact that for these games there is no incremental progress measure, so that we generate duplicate states in different layers and expand them several times. This results in more BFS layers (56 layers with 162,591 states opposed to eight layers with 25,734 states for Chomp and 63 layers with 1,866,488 states opposed to five layers with 129,776 states for Nim), which in turn results in more effort during the solving stage.

For the larger games the refined algorithm clearly outperforms the existing one. In all these games, an incremental progress measure can be found explicitly (e. g., a step counter in Catch a Mouse) or implicitly (e. g., the number of stones removed from the board in Clobber). Sheep and Wolf is the only game we solved, for which the second definition of the incremental progress measure is needed: Whenever the wolf moves, the progress does not increase, while the sheep can only move forwards. Thus, the sum of the rows of the sheep is a possible incremental progress measure. Due to the partitioning according to the layers, the BDDs stay smaller, and the image thus can be calculated faster. We also save time as we do not need to calculate the strong pre-images.

Algorithm 15.3: Finding all belief states.

Input: General Game, set of belief states of last step BS_{i-1}, set of observations of last step Obs_{i-1}.
Output: Set of belief states of this step BS_i.
$BS_i \leftarrow \emptyset$ **for all** $bs \in BS_{i-1}$ **do**
  determine all possible joint moves JM
  for all $jm \in JM$ **do**
    $Obs' \leftarrow observe(bs, jm)$
    if $Obs' = Obs_{i-1}$ **then**
      $bs' \leftarrow succ(bs, jm)$
      $BS_i \leftarrow BS_i \cup bs'$
return BS_i

15.4 Handling Incomplete Information

When playing incomplete information games, a player is confronted with the problem that it does not know the precise state. A first idea to get as much information as possible concerning the current state might be to evaluate all the observations based on the *sees* rules it has received during play. However, evaluating the corresponding rules is expensive and often does not yield enough information.

Another approach is to handle sets of *belief states*. A belief state is a state that might be true in the current state, but we do not know which of the entire set of belief states is the current state. We follow two approaches: First, to always store the full set of belief states, and second to store only a subset of the belief states and update it after a move or whenever we find that some belief state cannot hold anymore.

15.4.1 Full Set of Belief States

At the beginning of each playing phase, we must generate the new set of belief states BS_i based on those of the previous step BS_{i-1} (cf. Algorithm 15.3). For each belief state $bs \in BS_{i-1}$ we determine all the moves of all other players –we of course know our own move– and calculate all possible joint moves JM. At first, for each of these joint moves we check if the observations we would have achieved if these were the actual moves played equal those we did receive (Obs). If they do we know that this joint move was possibly performed, so that we calculate the corresponding successor state bs' and add it to BS_i.

While at first glance it seems great to have the full set of belief states to enhance the performance of the player, it comes at a great cost. Take the game of Poker with three players. Each player knows only the five cards it owns, so that there are $\binom{47}{5,5,37} = 1{,}304{,}872{,}821{,}252$ possible states. Storing all of these and efficiently operating on them is very expensive and the calculation of the next move might take more than the available play clock. Thus, for more complex games it seems better to store only a subset of all belief states.

15.4.2 Subset of Belief States

If we do not want to store the full set of belief states, we need to store a tree that allows us to find new belief states when we have to discard impossible ones. We call this tree the *belief state tree*. Each node of the belief state tree corresponds to a belief state, each edge to a joint move. For each node of legal belief states, we store the full set of possible successor belief states. These can have three values.

Algorithm 15.4: Depth-First Belief State Search (DFBSS)

Input: General game, belief state tree BST, set of observations of all steps Obs, current step i, size of belief state set $size$.
Output: Updated belief state tree BST, current set of belief states BS_i'.

```
layer ← 0
bs ← root(BST)
BS_i' ← ∅
while |BS_i'| < size or not (hasMoreLegalSuccs(root(BST))
or hasUnknownSucc(root(BST))) do
  jm ← jointMoveTo(bs)
  if isUnknown(bs) then
    Obs' ← observe(bs,jm)
    if Obs' = Obs_layer then
      markLegal(bs)
    else
      markIllegal(bs)
      bs ← parent(bs)
      layer ← layer − 1
  if isLegal(bs) then
    if layer = i then
      BS_i' ← BS_i' ∪ bs
    else if hasMoreLegalSuccs(bs) then
      bs ← nextLegalSucc(bs)
      layer ← layer + 1
    else if hasUnknownSucc(bs) then
      bs ← firstUnknownSucc(bs)
      layer ← layer + 1
    else
      if allSuccsIllegal(bs) then
        markIllegal(bs)
      bs ← parent(bs)
      layer ← layer − 1
return BST, BS_i'
```

legal We have evaluated that belief state and found that the observations we get when performing the ingoing joint move equal those we really observed.

illegal We have evaluated that belief state and either the observations when taking the joint move leading to it do not match the real ones or all its successors are marked as illegal.

unknown We have not yet evaluated that belief state and thus do not know if it is legal or not.

In order to find a subset of the possible belief states after a move was performed, we distinguish two approaches, one based on depth-first search, which we call *depth-first belief state search* (DFBSS), and the other based on random choice, which we call *Monte-Carlo belief state search* (MCBSS).

15.4.2.1 Depth-First Belief State Search (DFBSS)

Starting at the root of the belief state tree, i.e., layer 0, we continue in a depth-bounded DFS manner until either the current layer i contains the desired number of belief states ($size$) or the full tree has been evaluated (cf. Algorithm 15.4).

When we reach a node with unknown value, we evaluate it. If it is legal, we continue further along that node. Otherwise, we mark it as illegal and continue with its siblings.

Algorithm 15.5: Monte-Carlo Belief State Search (MCBSS)

Input: General game, belief state tree BST, set of observations of all steps Obs, current step i, size of belief state set $size$.
Output: Updated belief state tree BST, current set of belief state BS'_i.

```
layer ← 0
bs ← root(BST)
BS'_i ← ∅
while |BS'_i| < size do
  jm ← jointMoveTo(bs)
  if isUnknown(bs) then
    Obs' ← observe(bs, jm)
    if Obs' = Obs_layer then
      markLegal(bs)
    else
      markIllegal(bs)
      while isIllegal(bs) do
        bs ← parent(bs)
        if allChildrenIllegal(bs) then
          deleteAllChildren(bs)
          markIllegal(bs)
      bs ← root(BST)
      layer ← 0
  if layer = i then
    BS'_i ← BS'_i ∪ bs
    bs ← root(BST)
    layer ← 0
  else
    bs ← randomSucc(bs)
    layer ← layer + 1
return BST, BS'_i
```

When we reach a node representing a legal belief state we continue either to the first legal successor we have not visited in this search, or –if there is none– to the first successor with unknown value.

Upon reaching the current layer and evaluating the reached belief state as legal we store it in BS'_i and continue with its siblings.

15.4.2.2 Monte-Carlo Belief State Search (MCBSS)

A disadvantage of DFBSS is that it often has to evaluate large parts of the belief state tree. Especially when at some layer there were more belief states than we want to store and later a layer is reached where the full set of belief states is reduced so that it does not contain more than the number we wish to store anyway, DFBSS has to search the entire tree. Thus, the main bottleneck of DFBSS in practice is that it still is too slow. To overcome this problem, we use Monte-Carlo search in the belief state tree (cf. Algorithm 15.5).

Instead of performing depth-first search here we use several Monte-Carlo runs, each starting at the root node. When a state is reached that is marked as legal, we randomly choose one successor and continue from that. If a state with unknown value is reached, we must evaluate it. If it is illegal we mark it as such and recursively remove it and its predecessors if those now only have illegal successors. Afterward we restart at the root node.

The algorithm stops when the specified number of belief states (*size*) is found. Note that these states do not have to be different: otherwise, we might run into the same problem as with DFBSS, because we would have to evaluate the entire belief state tree if the full set of belief states is smaller than *size*.

While in most cases the MCBSS approach is a lot faster than the DFBSS approach, the memory consumption tends to be greater. In DFBSS only those nodes leading to legal belief states in the current layer remain, while in MCBSS we often store paths that are not fully evaluated to the current layer but rather ended in one state being illegal.

15.4.2.3 Weighted MCBSS

A problem of MCBSS (as well as DFBSS) is that the subset of belief states for the current step BS_i' does not necessarily contain the true current state. Furthermore, it might even be that all the states in BS_i' share only very few fluents with the true state.

To overcome this problem, we can use the rules for determining the opponents' rewards. The (average) rewards determined for the various belief states are stored together with the belief states within the belief state tree.

Instead of choosing a successor node uniformly at random, *weighted MCBSS* uses weighted probabilities. Thus, a belief state with higher stored reward values for the opponents will be chosen with higher probability than one with lower reward values. The idea here is that the opponents would typically perform a move that will ensure a higher reward in the end. The probability to choose a move is given by

$$P(\{m_{1i_1}, m_{2i_2}, \ldots, m_{pi_p}) = \prod_{k=1}^{p} \frac{estimatedReward(m_{ki_k})}{\sum_{j=1}^{n_k} estimatedReward(m_{kj})},$$

with p being the number of players, m_{ki_k} the move chosen by player k, $estimatedReward(m_{ki_k})$ the estimated reward for player k when choosing move m_{ki_k}, and n_k the number of possible moves of player k. Note that for us the chosen move is known, so that, assuming we are player x and have chosen the yth move, the estimated rewards for our moves can be set to 1 for move m_{xy} and to 0 for all moves m_{xz} with $z \neq y$.

15.4.3 Choosing a Move for a Set of Belief States

No matter if we manage the full set of belief states or only a subset, each belief state of the set can be seen as a classical GDL game and thus be handled by classical GGP approaches such as minimax or UCT.

To determine which move to choose the results of the games must be combined. For a minimax-based approach it is the move maximizing

$$eval(m) = \frac{\sum_{bs \in BS_i} reward(bs, m)}{|BS_i|},$$

with m being a legal move and $reward(bs, m)$ the estimated reward for move m in belief state bs.

For a simulation-based approach the same function might be used. However, it is possible to improve by integrating the number of simulation runs into the evaluation:

$$eval(m) = \frac{\sum_{bs \in BS_i} reward(bs, m) \times N(bs, m)}{\sum_{bs \in BS_i} N(bs, m)},$$

with $N(bs, m)$ being the number of times move m was evaluated in belief state bs.

In other words, the evaluations are weighted by their reliability. The results that were evaluated more often are weighted higher than those evaluated only rarely.

15.5 Experiments

Not many GDL-II games incorporating incomplete information have been published, and often they are either too simple or too hard. We selected the known benchmark Monty Hall and added the two games Memory and Stratego for evaluation.

Fluxii uses the full set of belief states. It has been added to the Fluxplayer infrastructure. During the startup time static analyses are conducted to improve the performance during the search. Once the set of belief states is computed, UCT is applied. Afterward, the results are merged. Nexusbaum applies the above explained approach of maintaining a subset of the belief states. The setup of the player including the calculation of the belief states and the accumulation of the results is implemented in *Java*, while for performance reasons the Monte-Carlo Simulations are implemented in `C++`.

The Monty Hall problem received much interest when it was first proposed. In that game we have a show master (here modeled as the random player) and a player who has to decide which gate to take. The show master places a car behind one gate and goats behind the other two. After the player has chosen a gate, the show master opens one of the unchosen gates containing a goat and allows the player to switch to the other unopened gate. This switching is the best move the player can take.

In Memory alias *Concentration* the random player deals eight cards. In case a player gets lucky, four moves suffice. However, the general best case is eight moves, which results in the full 100 points.

This version of Stratego (*orig. L'Attaque*), is played with six pieces for each player on a 3×6 board (cf. Figure 15.3). Pieces are removed according to fixed precedence rules, similarly to the original game. The goal is to take the opponent's flag before running out of steps. Otherwise, both players receive 50 points.

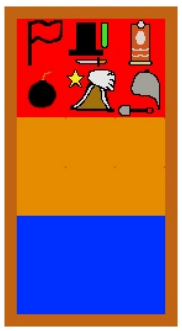


Figure 15.3: Stratego from a player's view.

The results are shown in Figure 15.2. Monty Hall: 100 games played, *belief state size* 10, *start clock* 10 seconds, *play clock* 20 seconds match the theoretical knowledge that a value of 2/3 can be achieved in optimal play (by changing according to the given knowledge). Memory: 100 games played, *belief state size* 10, *start clock* 10 seconds, *play clock* 120 seconds, show that almost optimal play of at most eight moves has been achieved. Stratego: 50 games each with exchanging roles and setting. Results for games against random with *belief state size* 10, *start clock* 30 seconds *play clock* 60 seconds in

Table15.3 shows the results of Nexusbaum against Fluxii with unweighted/weighted Monte Carlo Belief State Search (MCBSS/WMCBSS), and with Weighted Belief State Search plus UCT (WMCBSS+UCT) with 10 seconds start clock and 60 / 90 seconds play clock. While the first are still unfortunately for Nexusbaum, it went superior (56%/63% wins) when using weights and UCT.

Player	Games	Total Score	Average Score
Monty Hall	100	6700	67.0
Memory	100	9500	95.0
Stratego	50	4950	99.0

Table 15.2: Results in Monty Hall .

belief states	MCBSS	WMCBSS	WMCBSS+UCT
10	35.0	38.0	56
10	25.0	43.0	63

Table 15.3: `Fluxii` against `Nexusbaum` in Stratego with 60 (top) and 90s (bottom) without and with weightedness and with UCT.

15.6 Summary

The challenge of playing general games is to program autonomous agents that can play on a high level.

For GDL we presented an algorithm for solving general two-player turn-taking games making use of the information of the forward BFS. This brings the advantage that we do not have to use any strong pre-images, as all the successors of a given layer are solved for sure once this layer is reached. One shortcoming is that the BFS is mandatory, while this was not the case for the existing algorithms. Furthermore, it does not perform any duplicate detection, so that in some games more BFS layers are generated, and states are expanded multiple times.

One of the advantages is that we can stop the solving at any time and restart with the last partially solved layer later. Also, we can use the information we find on the hard disk as an endgame database, e.g., in combination with a general game player that uses UCT for finding good moves.

An interesting side-remark is that this approach can in principle also be used for any turn-taking game. All we need is the way to pass through the p-dimensional matrix of (possible) reward combinations, which gives us an opponent model. Unfortunately, this is not found trivially. Especially, in general game playing the agent gets no information as to which other agents it plays against, so that learning such a model seems impossible so far. If we assume that we can get an opponent model, we are able to solve all turn-taking games under the assumption that the model holds. The result is then similar to that of the Max^n algorithm, with the same shortcomings that algorithm has –namely, if one of the players does not play according to the model, the solution might be misleading.

For GDL-II that includes incomplete-information games we provided a competitive full-fledged GDL-II player, for which besides parsing, game controlling and some efficiency tricks, comes with a game engine for handling belief states.

Handling randomness and incomplete information is computationally hard. Even for single-player general games that can be cast to contingent action planning problems it is known that complexities rise drastically. Nexusbaum maintains sets of belief states and operates best with Monte-Carlo belief state search integrated into a UCT-based player. It plays single-agent games like Monty Hall and Memory almost perfectly, and —in the complex game Stratego— it outperformed Fluxii.

15.7 Bibliographic Notes

General games address the ultimate goal of generating intelligence, being able to deal with peviously unknown situation. In the last decades there is considerable progress in defining and solving general games.

The rules of games in the international game playing competitions [287] are given in GDL [459]. Alternatives to the logical formalism of GDL used in AI planning and in the description for general video games.

In the first two GGP competitions approaches making use of *Minimax* search [635] were prevalent, such as Cluneplayer [130] and Fluxplayer [558], the winners of the first two international GGP competitions. Since 2007, the most successful GGP agents such as CadiaPlayer [255] or Ary [472] use *UCT* [408] to calculate the next move to play.

As an example of the portfolio of UCT search control heuristics the *killer heuristic* [10] is a method originally introduced to Minimax algorithms to improve pruning. The list of moves is sorted according to the success of pruning. This can be ported to UCT by sorting the moves according to the statistics of rewards similarly to [256]. However, the effect is not always as big as in Minimax. Clobber has been considered in [11]. Stratego has been brought to superhuman performance with a deep learning algorithm called *DeepStack* [512].

GDL-II [611] (for GDL with incomplete information) became part of the accepted standard for general game playing. In [559] it is shown that GDL-II can be mapped to situation calculus, while [546] shows connections of GDL/GDL-II and epistemic logic.

Some players have been developed since, but so far documentation is rare. In the few international competitions on general game playing that supported incomplete information games in GDL-II, Fluxii by Stephan Schiffel was best. The name indicates that the player is based on the internationally very successful Fluxplayer [558]. Monty Hall has been discussed in [637]. Ludii is another general game system designed to play a wide range of games.

Chapter 16
Multiagent Systems

This chapter considers multiagent systems and their simulation. It documents the outcome of a study for addressing last-mile connectivity within a multiagent simulation system. We report on the simulation setup and the outcome in form of a feasibility assessment. The study provides a description of the agent model and its routing infrastructure as a step towards a rich model of the interactions that happen in urban mobility. We implement a scenario starting on a higher level of abstraction, drilling down to a running program. The multiagent model is generic in the sense that different transport agencies and vehicles can be added. It integrates planning with execution, a hot research topic these days. We will encounter that a sequence of calls to Dijkstra's single-source shortest-paths algorithm is crucial for planning and use an efficient implementation with radix heaps.

16.1 Introduction

Under[1] the umbrella term of smart mobility, the development of forward-looking traffic concepts for fast-growing metropolitan areas such as Bangalore, India has attracted considerable interest of regional authorities, transport planners and research. Important aspects include, amongst others, green goals like increased capacity utilization for transport modalities or reduction of carbon footprint, but also consumer-oriented goals such as safe, comfortable, and cost-efficient individual mobility. In this context, optimized use of existing and, potentially, planned traffic infrastructure, using new mobility concepts such as bike/car sharing and new ICT technologies as enabler, has come in sharp focus.

In pursuing the stated objectives, optimization potentials need to be determined on the part of individuals that utilize available offers for multimodal urban mobility, as well as on the part of transport providers. To procure well-founded assessments of the impact of particular optimization efforts time- and cost-efficiently, it is necessary to first design and implement a suitable simulation model of the as-is state of urban passenger traffic.

In the context of a study on opportunities for novel last-mile connectivity concepts, this study reports on a multiagent-based simulation model for the Bangalore urban region.

Multiagent-based simulation has been chosen as modeling technique over alternatives such as system dynamics, since the considered scenario allows for a natural mapping of traffic stakeholder to software agents. In addition, the approach allows for fine-grained modeling of individual traffic participants, which is considered essential for realistic simulation results. We present the simulation model with the realized software agents and discuss performance optimizations in applied routing algorithms which make the application of the simulation in the larger context of system optimization practical.

[1] This chapter is based on joint work with Christoph Greulich, Niels Eicke, Max Gath, Tobias Warden, Malte Humann, Otthein Herzog, and T. G. Sitharam. It puts together and improves the work from [200, 305, 306, 307].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_16

16.2 Multiagent-based Simulation

In multiagent systems, decision making is shifted from central, hierarchical planning and controlling entities to decentralized, heterarchically organized, autonomously acting software agents. These agents can represent objects, e.g., shipments, persons, taxis and buses. They act as individuals or are clustered in groups and can interact with each other by the use of negotiation and communication mechanisms. The general problem is split into smaller problems that agents solve locally in parallel within short time windows to optimize the behavior of the overall system. In cooperating systems, the agents' goal is to perform a globally optimized behavior and achieve common goals whereas in competitive systems each agent acts selfish to reach its own objectives.

Multiagent-based simulation (MABS) combines concepts of multiagent systems and simulation. A multiagent simulation system can combine distributed discrete-event or time-stepped simulation with decision-making encapsulated in agents as separate and concurrent logical processes. In classical simulation systems, the involved logical processes as well as interaction links have to be known in advance and must not change during simulation. This is not the case in multiagent-based simulation systems, as each agent may interact with all other agents. Agents may join or leave simulation during execution, e.g., depending on a stochastic simulation model or human intervention.

We use a client-server architecture. The system including the server itself can be distributed to several machines. This enables running numerous agents on many computers in parallel. Furthermore, knowledge about relations and properties of objects is encoded in ontologies. In general, the architecture consists of the following components: world model, physical objects, infrastructure, behavior definitions and agents. The agents in a scenario have methods for, e.g., sending and receiving messages as well as executing actions within defined behaviors. The agent is a logical entity that represents either an environmental process, an organization, or a set of physical objects.

The (physical) simulation world model of a scenario can be modeled as a graph so that the infrastructure can be mapped accordingly. Graph nodes represent, e.g., traffic junctions. Graph edges represent roads, rails, waterways, etc. To model real transport processes, we simulate scenarios within real traffic infrastructures that are imported from OpenStreetMap.

The client (GUI or console) starts simulation scenarios and provides information about simulation progress. The graphical user interface allows us to configure simulation parameters and shows simulation entities on a 3D world map. The command line client for console is for advanced users that are only interested in simulation results. Performance indicators are logged to a PostgreSQL database.

Of particular importance is the classification of agent types and the association of these software agents with entities in the simulation environment. The adopted modeling approach conceptually introduces a partition of all software agents in simulation into distinct agent communities, namely simulation actors and environmental agents. The former represent services or physical objects in order to evaluate the global performance, patterns of interaction or the design of particular agents. These agents act as artificial autonomous decision makers on behalf of their associated entity. Environmental agents manipulate the world directly, e.g., they are dynamically creating additional agents over the whole course of a simulation run.

16.3 Simulation Model

In the following we introduce the agent models that we have designed. We will see that the agents consist of *behavioral states*, which have a one-to-one correspondence to the actual Java implementation of the agent. We distinguish between simple behaviors (like *Init* or *HandleInformationRequests* or *Driving*) and complex behaviors (like *TransportPassengers*) that itself consist of an arrangement of behavioral states (see Figures 16.1 to 16.3). Complex behaviors are called Finite-State-Machine (FSM) behaviors. FSM behaviors are labeled tran-

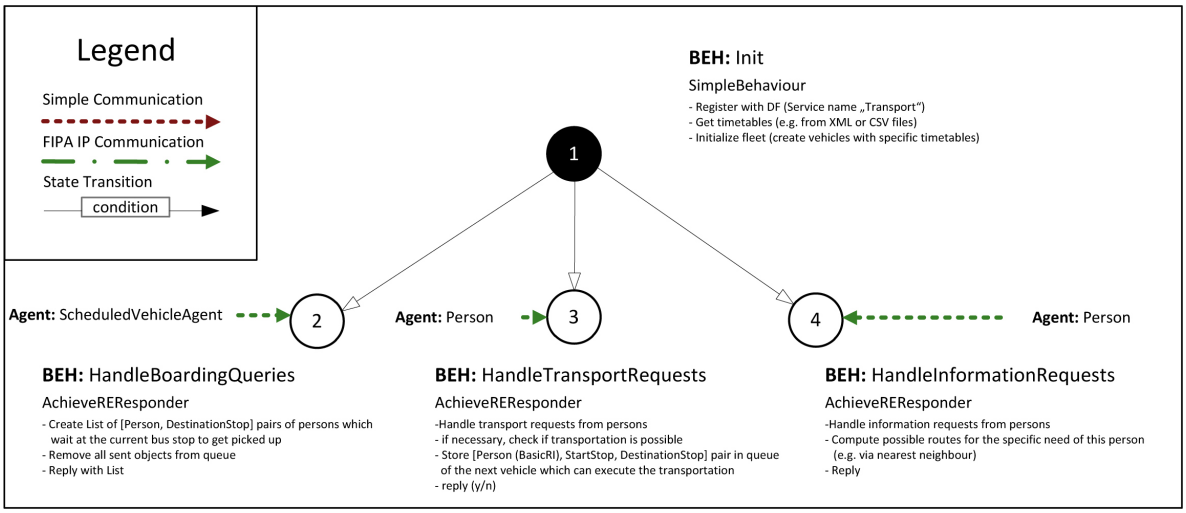


Figure 16.1: Model of the *Scheduled Transportation Company Agent*.

sition systems with a starting state (indicated with an incoming arc), some terminal state (encircled node), and state transition governed by some additional conditions.

Figures 16.1 to 16.3 also include communication arcs (dashed) that show whether or not a behavior is communicated with another agent. Communication parameter (e.g., Perf:Inform) on a communication arc refers to the performance of a message that starts an interaction protocol. Additionally, we have indicated which agent poses shortest path queries.

The *Scheduled Transportation Company Agent* is an implementation of a transportation company that provides an information infrastructure for answering initial transport queries (HandleInformationRequests), sets up and maintains a list of persons to be transported at stops for a specific vehicle (HandleBoardingQueries), and receives, answers, and commits transport requests of persons at stops (HandleTransportRequest).

More formally, a transport request of a person wrt a routing graph infrastructure G with node set V is assumed to consist of a start location $v_0 \in V$ and an end location $v_* \in V$ of the travel. Moreover, a time stamp can be associated to the travel, which in the simplest assumption is the current simulation time. The answer of the scheduled vehicle agent is a selection of transport options that each contains the first stop $h_0 \in V$ to be picked up and the last stop $h_* \in V$ of the transport. If $h_0 \neq v_0$ we have that there might be remaining efforts needed for the human to reach the stop and if $h_* \neq v_*$ we have that remaining efforts are needed to reach the final destination of the trip.

It is possible that the trips offered by the multi-vehicle transport agency consist of different transportation options (bus, metro). In the first approximation, we only consider buses. In complex settings it is possible to recursively plan the prefix and suffixes of a trip with a different transport agent.

Its agent model is shown in Figure 16.1. The agent is initialized once the scenario is started and starts with the behavior Init (matching to all other agents). After registration with the system-wide directory facilitator and acquisition of time-table information from a configuration file, the agent is responsible for invoking time-dependent initialization of *Scheduled Vehicle Agent* that acts on behalf of the fleet. The three sub-behaviors are spawned, and run in parallel.

The *Scheduled Vehicle Agent* receives on its initialization a schedule from the agency that it has to follow on a daily basis. The timetable information contains the arrival and departing time of each stop in the tour. If a vehicle is running late, it usually tries to catch up with its schedule, reducing waiting times. In case of a bus company this will be a bus. Other examples are trains and metros. Its model is shown in Figure 16.2. The FSM behavior for scheduled driving mainly implements a cycle on boarding, debarking, and moving. Furthermore, it uses an interaction protocol to communicate with the transportation company agent to receive new instructions.

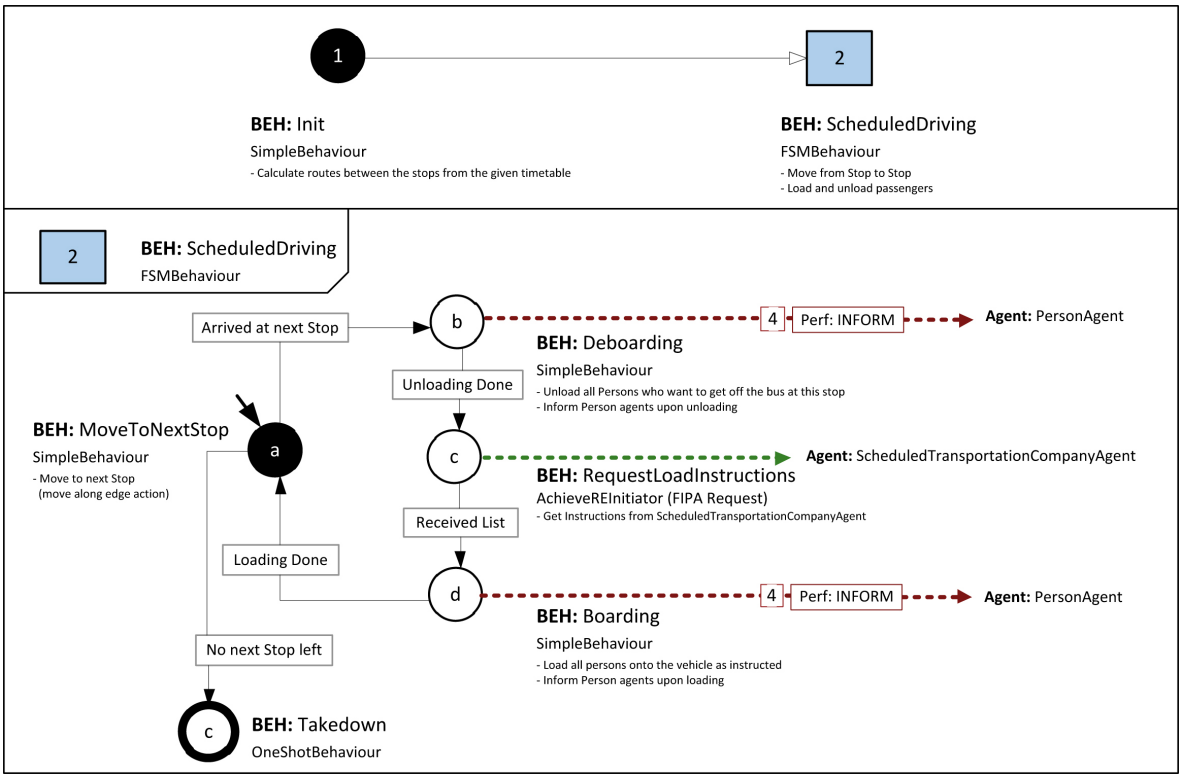


Figure 16.2: Model of the *Scheduled Vehicle Agent*.

The *Autonomous Transportation Vehicle Agent* is a complex agent that allows us to sign contracts with customer and to follow routes. It combines the functionality of a scheduled transportation agent and the scheduled vehicle agent. An example is an individual taxi or an individual rickshaw. On the top level of the specification, the autonomous transportation vehicle agent has three states. At first, it is invoked and it receives its timetable. In contrast to the transportation agency, it does not handle boarding queries on the top level to maintain a list of persons, but immediately and individually executes transport requests. The transport passenger FSM behavior is similar to the scheduled driving in terms of boarding, debarking the passenger, but does not iterate over a sequence of stops. The answer of the scheduled vehicle agent to a transport query (v_0, v_*) usually is one singleton transport option with the human to reach the first stop and no efforts to reach the final destination of the trip.

The *Birth-Giver Agent* initializes person agents with several parameters like the start location, a certain budget in time and cost, as well as a target location. All values are random numbers, drawn according to a given probability distribution. Optionally, the start and end location can be specified manually by determining a fixed node. The simple model with only one state that iteratively creates person agents for the simulation. The realistic modeling of the random process is crucial for the applicability of the simulation outcome. The better the model the better its prediction. These data might be indirectly inferred by information on where people live and where they work, or by monitoring their current use of vehicles. At the end, a rather complex probability distribution for transport requests, humans and their queries should be derived. To determine the next bus stop for a transport request, a nearest neighbor search must be conducted.

The *Person Agent* is a complex agent. It communicates with transportation agencies as well with vehicles, that transport the represented human. It plans and executes routes. A person is also able to actively execute the transportation task without any vehicle by walking. The implemented model is shown in Figure 16.3. We see a hierarchy of complex FSM behaviors. Moreover, it shows that a person lives in a loop of planning and execution of the plans. Based on the dynamics in the simulation system plans can fail and require re-planning.

For the generation of plans (planning behavior) shortest paths are generated for walking, as well as transportation agencies have to be contacted. While the start and goal locations are assumed to be on an arbitrary node in the map, not all transportation requests can be satisfied by bus, so that planning does include padding vehicle usage with walks to or from the stops of the vehicles. Once the plan is fixed, it goes to execution (ExecutePlan behavior). For such execution of plans we see that there is a distinction between active behaviors like walking and passive behaviors for being transported (while riding a bike might be attributed as an active option in real life, in this model it can also be view as a passive behavior in form of a rental service with a contract that must be sealed). The starting point of the behavior looks at the next step in the plan and decides whether it is an active or a passive behavior. We see that passive transport has to be monitored and can fail, so that the planning behavior has to be reinvoked on termination. The persons are removed from the simulation once they reach their final destination.

16.4 Shortest Path Search

Dijkstra's shortest paths search is again realized using memory-efficient joint representation of graph and radix heap nodes. The code for radix heaps is shown in Program 16.1. The radix heap assumes that all edge costs in the graph are integers bounded above by C. The result is that Dijkstra's algorithm can be implemented with a time complexity of $O(m + n \lg C)$, where n is the number of nodes and m is the number of edges. Given that the logarithm of a 64-bit integer is bounded by a constant $\lg C = 64$, the running time on a modern computer is linear $O(m+n)$. If edge weights are doubles, $\lg C = \lg(1.79769 \cdot 10^{308})$ is also bounded by a constant. Moreover, assuming that two pointers for linking the elements in the heap are already provided with the nodes, the memory needed for the buckets is $O(\lg C) = O(1)$.

16.5 Experimental Setup

To perform the simulation for Bangalore we extracted the route navigation data from the OpenStreetMap. As Bangalore was not predefined as a coherent district, we defined a bounding box on the city and included also streets that cross the boundaries instead of clipping them exactly at the border of the bounding box. The GPS/GIS data on the road infrastructure of Bangalore was exported to a database, that can be included directly. The road infrastructure contains 49,399 nodes and 134,222 edges and includes the International Airport of Bangalore (BIAL).

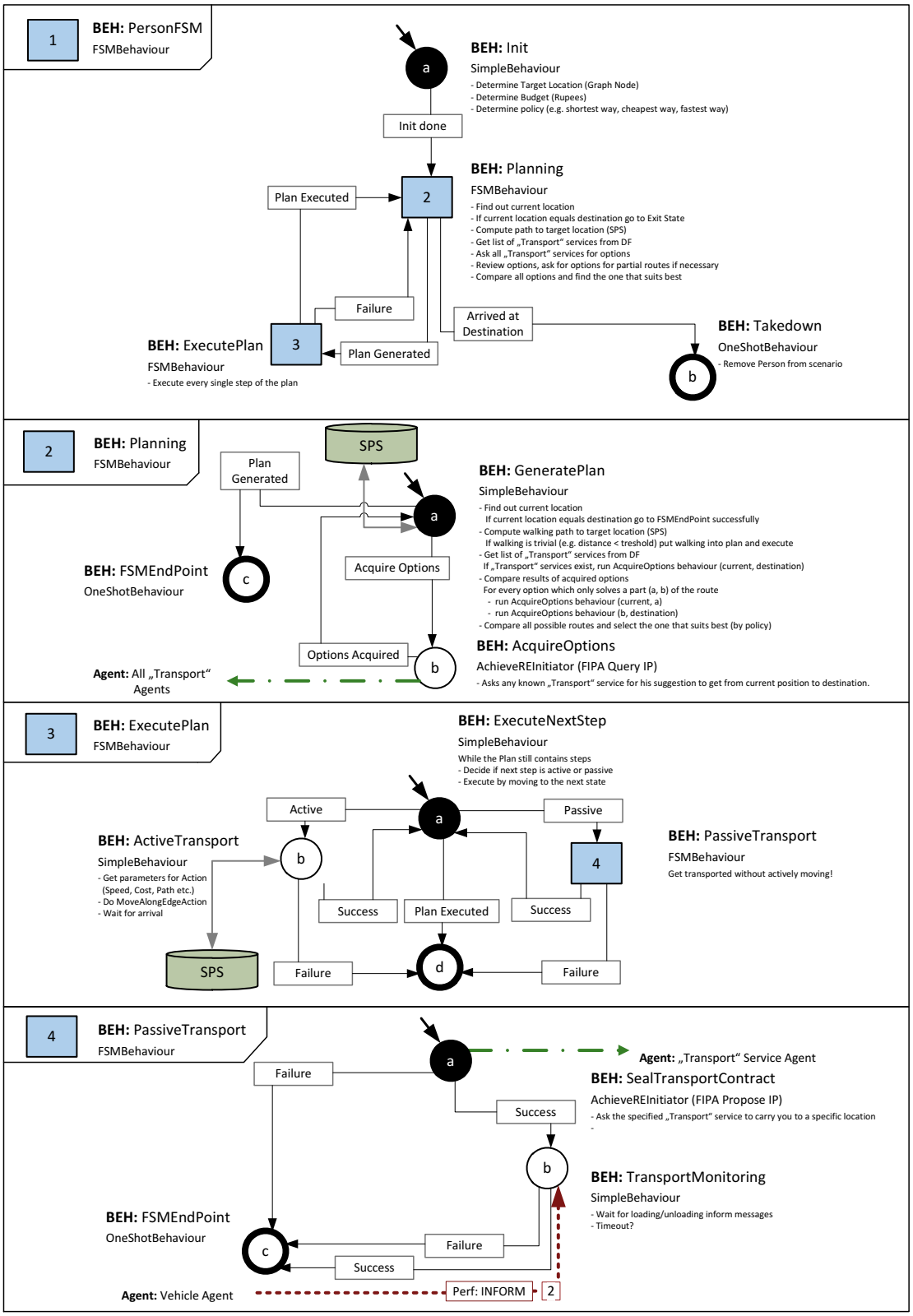


Figure 16.3: Model of a *Person Agent*.

Program 16.1: Program for radix heaps.

```
public class Radix {
  long S, n, u[], b[]; int B; Node buckets[];
  public Radix() {
    S = Long.MAX_VALUE; B = (int) (Math.ceil(Math.log(S) / Math.log(2))) + 2;
    buckets = new Node [B]; b = new long [B]; u = new long [B];
    for (int i = 0; i < B; i++) { buckets[i] = null; b[i] = u[i] = 0; }
    b[0] = 1; b[B-1] = Long.MAX_VALUE;
    for (int i = 1; i < B-1; i++) b[i] = 1L << (i-1);
    u[B-1] = Long.MAX_VALUE; n = 0;
  }
  public long size() { return n; }
  public boolean empty() { return (n == 0); }
  public Node next(Node p) {
    if (p.succ != null) return p.succ;
    else {
      int next = p.bucket + 1;
      while ((next<B) && (buckets[next] == null)) next++;
      if (next == B) return null; else return buckets[next];
    }
  }
  public void insert_node(Node p, int i) {
    p.succ = buckets[i];
    if (buckets[i] != null) buckets[i].pred = p;
    p.pred = null; p.bucket = i; buckets[i] = p;
  }
  public void extract_node(Node p) {
    if (p.pred != null) { Node q = p.pred; q.succ = p.succ; } else buckets[p.bucket] = p.succ;
    if (p.succ != null) { Node q = p.succ; q.pred = p.pred; }
  }
  public void adjust(long m, int t) {
    int i; u[0] = m;
    for (i = 1; i < t; i++) { u[i] = u[i-1] + b[i]; if (u[i] > u[t]) break; }
    for (; i < t; i++) u[i] = u[t];
  }
  public int find(Node p, int i) {
    if (p.element == u[0]) return 0;
    while (p.element <= u[--i]); return i+1;
  }
  public Node top() { return buckets[0]; }
  public Node insert(Node p) {
    long k = p.element;
    if (n > 0) insert_node(p,find(p,B-1)); else { adjust(k,B-1); buckets[0] = p; p.bucket = 0; }
    n++; return p;
  }
  public void decrease(Node x, long k) {
    x.element = k;
    if (k <= u[x.bucket-1]) { extract_node(x); insert_node(x,find(x,x.bucket)); }
  }
  public Node extract() {
    for (int i = 0; i < B; i++) {
      Node p = buckets[i];
      if (p != null) { extract_node(p); n--; return p; }
    }
    return null;
  }
  public Node extract(Node x) {
    int i = x.bucket; extract_node(x);
    if ((n > 1) && (i == 0) && (buckets[0] == null)) {
      int j = 1;
      while (buckets[j] == null) j++;
      Node p = buckets[j], d = p.succ;
      while (d != null) { if (d.element < p.element) p = d; d = d.succ; }
      adjust(p.element,j); extract_node(p); insert_node(p,0); p = buckets[j];
      while (p != null) { Node q = p.succ; extract_node(p); insert_node(p,find(p,j)); p = q; }
    }
    n--; return x;
  }
```

Table 16.1: Amount of bus commuters between BIAL and the city center.

Data Collection	BIAL to City Center		City Center to BIAL		
Day of Week	Single Ticket	Pass Ticket	Single Ticket	Bus Pass	Total
Sunday	2,954	349	2,528	566	6,397
Monday	2,251	523	2,061	1,220	6,055
Tuesday	1,902	508	2,151	1,430	5,991
Wednesday	1,715	452	2,079	1,273	5,519
Thursday	1,899	512	2,187	1,146	5,744
Friday	2,213	498	2,673	1,289	6,673
Saturday	2,502	362	2,668	711	6,243

```
# Line=BIAS-4
# Start_Times=07:00, 08:00, 09:00, 10:00, 10:30, 11:00, 11:30, 12:00, 12:30, 13:00, 13:30, 14:00, 14:30, 15:00, 15:30,
     16:00, 16:30, 17:30, 18:00, 18:30, 19:00, 20:00, 21:00, 22:00
JeevanBimaNaga_Stop | 00:01 | 00:01
ChinmaiahHospital_Stop | 00:04 | 00:01
CMHRoad_Stop | 00:05 | 00:01
TajResidency_Stop | 00:04| 00:01
LakesideHospital_Stop | 00:03 | 00:01
StJohnsRoad_Stop | 00:07 | 00:01
JCNagara_Stop | 00:05 | 00:01
MekhriCircle_Stop | 00:06 | 00:01
Hebbala_Stop | 00:04 | 00:01
JnOfKogiluCross_Stop | 00:03 | 00:01
```

Figure 16.4: An exemplary bus schedule.

The bus routes and associated information on travel fares as well as fleet sizes is supplied by the Bangalore Metropolitan Transport Corporation (BMTC) authorities. They refer to real-life data. We decided to focus on the last-mile connectivity of passengers at BIAL. The distribution of passenger transports measured for a particular week as shown in Table 16.1 illustrates that there are about 6,000 persons who use the bus lines on a day. In comparison there are about 10,000 persons taking a taxi (trips per day).

Information on real-life bus schedules as well as bus stops is added manually. The simulated scenarios contain eight bus lines, whose first or last stop is the BIAL. We specified the bus lines and their timetables within a simple text file that contains a sequence of starting times for buses operating on a line and the relative times from one bus station to the next. An exemplary bus schedule is shown in Figure 16.4.

The implementation of agents for the initial study is fully operational. We have person agents (Person), generator agents that samples humans and rickshaws (Birth-Giver), bus agents that follow timetables (Scheduled Vehicle), a bus agency agent (Scheduled Vehicle Agency), as well as bikes as a substitute for walking for longer distances. Taxis (Individual Transportation Vehicle) can be added on request.

Parameters to instantiated birth-givers are used to change the ontological concept and thus the outlook, speed, and other properties of physical objects). In our scenario, a first birth-giver agent is responsible for creating person agents that act for persons located in the inner-city district. The desire of this person is to arrive the airport as soon as possible. Person agents that are created by a second birth-giver agent represent individuals whose desire it is to get from the airport to the inner-city district. The number of generated agents is specified in Table 16.1. Finally, a third birth-giver agent generates a fixed number of rickshaws which start at the airport and also drive to the inner-city district.

To generate the start and end location randomly, we implemented a random walk strategy starting at a bus stop to generate requests at certain nodes with an arbitrary distance. Therefore, we ensure that transport requests are not far from the encoded bus stations. Simulating dynamically changing traffic conditions is the subject of further research. As a result, edge following is mainly determined by the speed of the human or the vehicle as well as the type of the road.

Figure 16.5: Running simulation with buses, persons, rickshaws and bikes.

Figure 16.6: Automatically extracted graph from OpenStreetMap.

Moreover, we added bikes, so that persons have the flexibility to decide how to get to the bus stop or home: if distance is close enough (smaller than some predefined threshold) they walk, otherwise they ride by bike (see Figure 16.5). The maximum walking speed of a person is limited to 5 km/h while the maximum speed of a bike is limited to 30 km/h. We simulated a time span of a whole week.

The scenario has been adapted to the infrastructure of the German city Bremen (see Figure16.6). The infrastructure has been extracted from OSM and timetables have been provided by the local public transport organization BSAG. Other cities can be used as long as they provide the same map timetable format.

Various sources for real-world data cause a graph matching problem. OpenStreetMap provides detailed information about the infrastructure and public transport network of a specific region, but user generated content may be erroneous or incomplete, recent changes to the real world may not be reflected yet, no timetable information about public transport networks is available. Public transportation companies provide operating data in standardized databases, but geographic information may be insufficient, and no infrastructure information is available. Therefore, itineraries of both data sets have to be matched.

The public transport and route planning graphs are mapped using a combination of GPS and string proximity. For the latter a dynamic programming algorithm for approximate string matching (extending the edit distance, similarly to the multiple sequence alignment solution of Chapter 20) has been implemented. The Needleman-

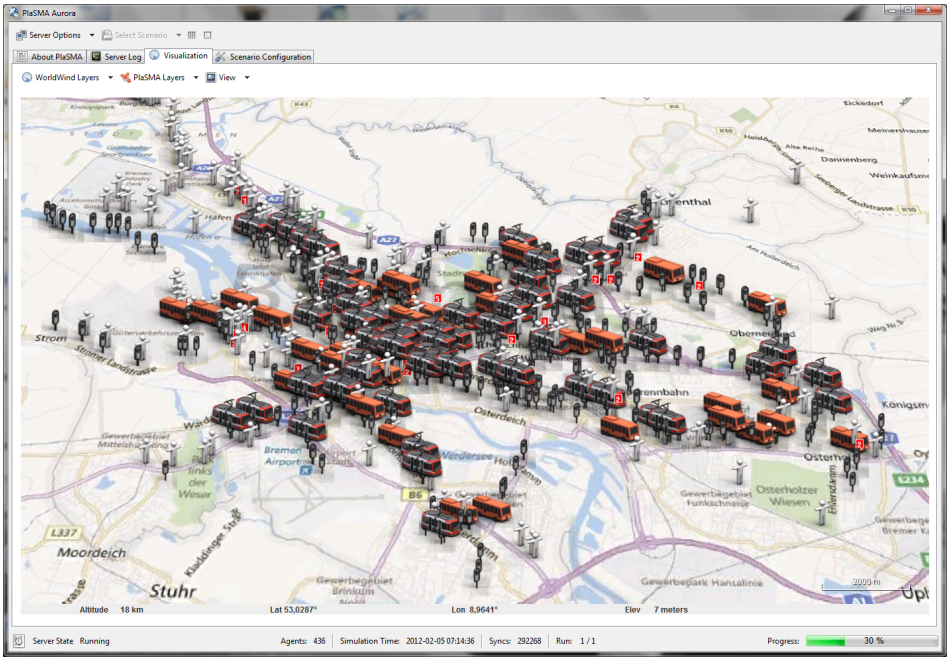


Figure 16.7: Running simulation of mixed public transport and route planning with buses, trams and walking persons.

Wunsch algorithm solves the global sequence alignment problems and detects gaps. Determining the similarity of two stops from different datasets. Names of stops are not guaranteed to be different due to abbreviations, etc. Comparison of names is an inferior global sequence alignment problem and the comparison of geographic coordinates of stops improves results if available.

A traffic information system was added to provide each commuter with a plan of stops, changes, and walking routes. The transportation company agent manages timetables for the fleet, answers individual information requests by potential passengers and maintains a list of waiting passengers per stop. The algorithms behind the scenes are variants of Dijkstra's method to compute shortest path in a time-dependent or time-expanded network.

We extended the system to solve the tourist traveler problem, a multi-goal version of the shortest public transport problem with time windows. Monte-Carlo search helped to find the visitor tours.

16.6 Current Status and Findings

The road infrastructure (including the international airport BIAL) is completely mapped to GIS and imported by the simulation system.

Using the PostgreSQL database interface, plots can be generated. For our sample scenario we are, e.g., interested in the number of passengers at the bus stops (see Figure 16.8). The total amount of passengers is shown in Table 16.1. No. 2 is the BIAL and, therefore, the place either of arrival or departure of each person. Low traffic is primarily caused by a high density of bus stops within a certain district (the simulation contains eight bus lines from or to the BIAL). In total we simulated more than 45,000 agents with a max. 1,100 agents running concurrently.

Performance indicators recorded in different simulations can easily be combined (e.g., values averaged). Besides the number of passengers that board or debark a bus, there are a host of other interesting performance

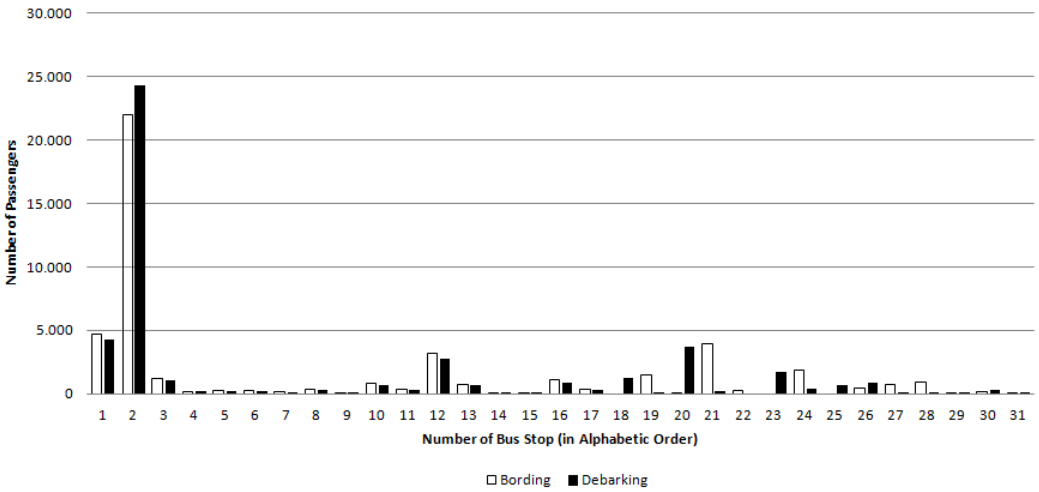


Figure 16.8: The open bars show the number of boarding and the closed bars the number of debarking passengers at each bus stop within a whole week. The bus stops are numbered consecutively to their official name in alphabetic order.

indicators for the simulation like the time spent by persons to reach their destination, the budget needed for the travel, or a combination of both. Experiments have shown that the bottleneck of the computations is computing several shortest paths on the routing graph by each person agent. Buses do not change their routes if there is no occurrence of unexpected events. Therefore, the computation time of the buses is minimized by the use of a cache.

The worst-case running times of applying the Dijkstra single-source shortest-paths algorithm (without terminating at the goal) are: Generating a graph with 100,000 nodes and 1,000,000 edges took 2,427ms, all pair shortest paths search 707ms; generating a graph with 1,000,000 nodes and 10,000,000 edges ran 30,574ms, all pair shortest paths search 34,631ms.

We see that searching all shortest paths slightly dominates the initialization. However, when terminating at goal states, the search time decreases significantly to 11,988ms. As a result, Dijkstra shortest paths search runs sufficiently fast even on large graphs. If better performance is needed, A* with Euclidean distance heuristic or bidirectional shortest paths search can be used. Moreover, there are speed-up techniques that trade preprocessing the graph for a better search time.

16.7 Summary

The challenges to be addressed in ongoing multi-disciplinary research on smart mobility in Indian metropolitan areas such as optimized utilization of existing and planned traffic systems, are manifold. This chapter reported on the status of a fine-grained multiagent-based simulation model for urban mobility, for experimentation within the simulation system. The physical simulation world model builds upon detailed OpenStreetMap data for the traffic infrastructure. It allows for the mapping of extensive public transport networks.

The generation of individuals that utilize modeled transport modalities can be configured according to actual distributions. Besides agent models for the modeling of transport operators such as the BMTC/BSAG and independent operators (using, for instance, taxis and rickshaws), a particular focus was put on the rational modeling of transport customers. These are equipped with capabilities for interleaved planning and execution of intermodal transport schedules. Experiments have shown that the multiagent simulation system handles a realistic number of passengers in a time frame which makes extensive experimentation practical, despite complex planning and route-finding calculations performed by the simulation actors.

We see this as an encouragement to advance the simulation model in different levels. On the infrastructure level, we seek to directly acquire data from publicly accessible web services operated by transport stakeholders to facilitate, amongst other things, modeling of active bus lines and congestion-induced delays. Further optimization to the planning and route-finding algorithms used by the agents will be implemented as discussed before to accommodate the desire to scale simulations towards larger parts of a traffic system. The multiagent system could be extended so that new transport modalities (e.g., taxi services) can be simulated. A particular focus is on the advancement of the planning competences and flexible troubleshooting behaviors for the person agents. For a transition to a rich model for simulated persons that comprises parameters such as financial constraints, constitution and individual preferences, which allow the specification of realistic groups of traffic participants, planning needs to accommodate these additions.

16.8 Bibliographic Notes

The multiagent simulation system is event-driven and has been designed to solve and evaluate scenarios in the logistics domain [644]. The system simulates processes, where planning and decision making are delegated to autonomous acting agents [564]. IT extends the well-known JADE framework [41] which implements the standards for agent interaction and communication defined by the IEEE Foundation for Intelligent Physical Agents (FIPA) [565]. It provides a discrete time simulation and ensures correct conservative synchronization with time model adequacy, causality, and reproducibility [283]. Based on the identification of important quality criteria for multiagent systems, presented in (Gehrke et al., 2008), a coordinated conservative synchronization approach has been adopted in the simulation system, which is handled concertedly by the simulation controller hierarchy.

The chapter refers to a user needs study [9] and the 2010 *Traffic Management Plans for Major Towns in Bangalore Metropolitan Region* for the Bangalore Metropolitan Region Development Authority conducted by Wilbur Smith Associates.

We have carried over techniques well-proven in autonomous logistics context such as planning under uncertainty and temporal constraints [457] or team formation as enabler for unlocking of attractive transport offers [564] (For instance, finding an agreement to share a taxi or rickshaw) to the urban mobility context. Spontaneous ride-sharing concepts can be considered as well as an option alternative [661]. Strategically, the rich simulation model that is under active development constitutes a versatile experiment platform for the evaluation of smart mobility concepts in large-scale urban areas like Bangalore. Complementary promising concepts include, amongst others, the integration of person agent functionality into a mobile phone-based intelligent travel assistant or the introduction of short-term car-, rickshaw- or bike- sharing solutions [570]. Here, multiagent-based simulation can be used for feasibility assessments and optimization of vehicle pickup and return sites.

Chapter 17
Recommendation and Configuration

In Chapter 11 matrix factorizations were studied. In this chapter we first look at recommendation and its relation to clustering. We explain how general the method is and how it attacks the curse of dimensionality. We review the problem of bitvector classification, going forward to clustering, and explain pathological behaviors of current algorithms have.

Next, we present the design and an implementation of a recommender system that supports the users' choice of parameters in an ongoing product configuration task. The machine learning approach that is extended to work for the configuration problem at hand is based on a combination of association rule mining and case-based reasoning in form of k-nearest neighbor search. The evaluation of the learning efficiencies is executed for a body of real-world data instances form industry. It shows a trade-off between achieving a highly correct prediction and a low misprediction rate.

17.1 Introduction

We[1] are living in a world of increasingly individualized products, an issue which has led to the marketing strategy of *mass customization*. In the logistics of smart factories, lot sizes 1 are a mainstay trend. With the increasing complexity of products, the problem of *product configuration* arises, which refers to the process of assembling a valid end product, given the large number and parameterization of different subcomponents for its assembling. Roughly speaking, a configuration is an assignment of values to parameters for product components.

To counter *mass confusion*, in the last decade, several commercial configuration tools have been developed that support the customers in the product configuration process. As product subcomponents interact, the main component in such software is constraint propagation, which –based on the given course of selections made so far– limits the number of future design choices.

We design, implement, and evaluate a system that, in cooperation with the existing configuration software and based on historical records, recommends certain properties of the products to be configured. Such a recommender system that learns users' preferences is the natural extension to constraint propagation. It applies machine learning to assist the configuration process.

We first introduce systems for recommendations and the potential problem of clustering binary inputs. Then, we look the process of (structured) product configuration. Next, we discuss basics of recommender systems for product configuration. In the sequel, we describe machine learning algorithms that we have selected, and the adaptations needed to have them work for the complex input of sample configurations. Last, but not least, we evaluate two orthogonal approaches, and —suggested by the empirical outcome— a hybrid of the two.

[1] This chapter puts together and improves pieces of unpublished joint work with Daniel Rietmüller and Björn Schwarze.

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_17

17.2 Recommender Systems

As one of the most successful machine learning applications in place, recommender systems facilitate an important technique to handle information overload. The hypothesis is that, if a person's preferences are known, such system can recommend products that the person *may* like. Recommender systems can help decide on a variety of items, such as movies, books, and restaurants. *Content-based* and *collaborative filtering* are commonly used to make those recommendations. However, there are more options such as demographic and hybrid filtering. Each system has several advantages for the customer and the company. Although the customer is offered a personalized service, the user is limited by his own knowledge. An individualized service can help the user to decide by reducing the amount of time spend on deliberate about the given options. Nevertheless, the company generates profiles about their user, enabling them to collect data about demographics, purchase history, and preferred interactions. This makes business analysis and recognizse market trends possible. Good recommendations make a shop more user-friendly and increase user's loyalty. As a result, the users are more likely to revisit the shop, since they can easily find articles of interest which leads to increases sales.

The dimensionality of the input data affects the speed of the classification and clustering as well, but it also often causes a numerical instability known as the *curse of dimensionality* (CoD). It is described as a general unreliability of distance computations for data sets with rising dimensionality.

With content-based filtering (CB), the description of the product is used to learn which preferences the user has. CB is rarely used, due to the increased adversity for machines to achieve the required *level of understanding* Specifically, it is very difficult in some areas of application to determine useful properties of the articles. However, the characteristics of an item are required for this procedure. The limitation of the available content of the products poses a problem for content-based RS, which can lead to over-specialization. The approach is also subject to further restrictions. For example, the system can only suggest items to the user that are very similar to the products already consumed.

The process of collaborative filtering (CF) is more widely used than CB. CF offers a simpler approach, by collecting a large amount of information on the behavior of different users. The CF collects a large amount of information about the behavior of different users. The most prominent example was implemented by the company Amazon. The purchasing behavior of various articles by several million people is logged in one central database. In this way, a vector can be created for each article, which describes the purchasing decisions of all customers. However, CF usually requires a huge dataset. CF finds an application in industries like surveillance data, mineral exploration, analyse large areas, financial data related to financial organization, electronic commerce, and various web applications.

Cold Start Problem. Currently, it is challenging to introduce new users or new items into an existing shop catalogue. Without previously collected data, the behavior of the new user cannot be predicted. Further, new items have not been rated or purchased yet. This obstacle is described as the *cold start problem* (CSP). Nevertheless, there are several ways to resolve this problem. For example, the novel user can be asked at the beginning how he would rate an article. Additionally, they could also provide information about his or her preferences. Also, assumptions can be made based on of their demographic data. Several different algorithms tried to solve the CSP. Combining CB and CF can result in the recommendation being of moderate quality, when it not possible to provide a description of a product.

Challenges with Rating Users. In general, a sparse evaluation in collaborative RS makes it difficult to make accurate predictions about products. Especially for cross-context collaborative RS, precise recommendations are dependent on multidimensional vectors, which is known as sparse evaluation. Reviews of so-called *grey sheep*, who have a different opinion about a product compared to most consumers, are problematic. An RS cannot benefit from such evaluations. To make the algorithms more efficient, those opinions can be filtered out to reduce their influence. Another problem is the malicious manipulation of product ratings in order to gain a competitive advantage. Additionally, this so-called shilling attack can diminish the credence of a RS. Generally speaking, it is difficult to make accurate predictions about products in collaborative RS terms. Especially in the case of cross-context collaborative RS precise recommendations remain enigmatic due to the use of multidimensional vectors.

Challenges with Clustering. The feature vector that combines various properties can be described as a point in hyperspace. Challenging aspects of clustering include that the hyperspace often has as many dimensions as the the vector has several properties. The number of properties alias the dimension of a vector has a strong impact. On one hand, if too few features were chosen, clouds of different classes could no longer be separated. On the other hand, if too many features were used, the number of states which can be represented would increase with each dimension. If the number of training data records does not increase as the number of dimensions increases, the classification results get deteriorated.

In addition to the number of dimensions, the type of characteristics is also important. For example, it is conceivable to use several properties in a feature vector that share mutual dependencies. Hence, it is inexplicable to process all characteristics together. Mathematically speaking, a smaller Z-dimensional space spans a larger R-dimensional space, where Z corresponds to the number of linearly independent features. Methods of dimensionality reduction aim to transform the R-dimensional space into a Z-dimensional space, while the content of the original information should be preserved.

The Pearson correlation coefficient is the most used method to find correlations. However, Pearson's coefficient is restricted to linear correlation. Spearman and Kendall are the two most popular non-linear correlation coefficients. These methods are limited by monotonous functional dependencies. Other known methods measure the correlation of random variables such as distance correlation, Hoeffding's independence test, Maximum Information Coefficient (MIC), Hilbert-Schmidt information criterion (HSIC), and Heller-Heller-Gorfine distance (HHG).

In the following, the variables X and Y describe undefined data records. With the *distance correlation*, random vectors can be tested for their independence. Since the Pearson correlation coefficient is limited to linear relationships, the distance correlation eliminates this deficiency. A correlation of zero by Pearson does not imply independence, only uncorrelatedness. A distance correlation of zero, however, says that random vectors are independent. The distance correlation is defined as $R^2(X,Y) = v^2(X,Y)/v^2(X)v^2(Y)$, if $v^2(X)v^2(Y) > 0$, and 0, if $v^2(X)v^2(Y) = 0$, where v is the *variance*.

The *maximum information coefficient* (MIC) describes an algorithm that searches a large data set for pairs of variables. The algorithm is designed for data sets with several hundred variables. For this, MIC calculates the correlation for each pair and arranges the pairs according to their score. Different types of functions, such as linear, exponential or periodic, are taken into account. The different function types with similar $R^2(X,Y)$-values are studied. If there is a relationship between two variables, they are displayed as a grid on a scatter plot, which allows the data to be partitioned. A $X \times Y$–lattice with the highest induced mutual information is found for each pair. For this purpose, all grids are examined up to a maximum grid resolution depending on the sample size. The result is the greatest possible mutual information that can be obtained through any $X \times Y$ grid applied to the data.

Typically, distance measures such as Euclidean, Manhattan or Cosine are used to calculate the distance between vectors. Due to the CoD, all distances of the vectors in high-dimensional hyperspaces are almost equal and orthogonal to each other. The use of distance measures as a degree of similarity in the height dimension space, therefore, easily leads to inaccurate clustering results. To avoid the problem, MIC procedure is used as an alternative. The MIC-k-means-algorithm is a k-means algorithm that uses MIC instead of the conventional Euclidean distance measure to determine the similarity between the vectors.

The main goal of *Nebula* is to group patients based on their medical similarities. To do this, the biomarkers must be identified to capture key features for each subgroup. Equipped with a non-parametric Dirichlet process mixture model, Nebula can learn interactively. The first challenge is to asses whether an individual can be assigned to a sub type based on a biomarker with significant confidence.

Figure 17.1 displays a taxonomy for recommender systems and proposed solutions. In summary, there are two different levels of problems. One problem level based on CB and CF is directly related to the recommendation; these are optimization problems that describe the behavior or the performance of the recommender systems. On the other hand, there is a level that describes general algorithmic problems with clusters in different dimensions. This problem level only has to do indirectly with recommender systems, but can also be transferred to other areas of application.

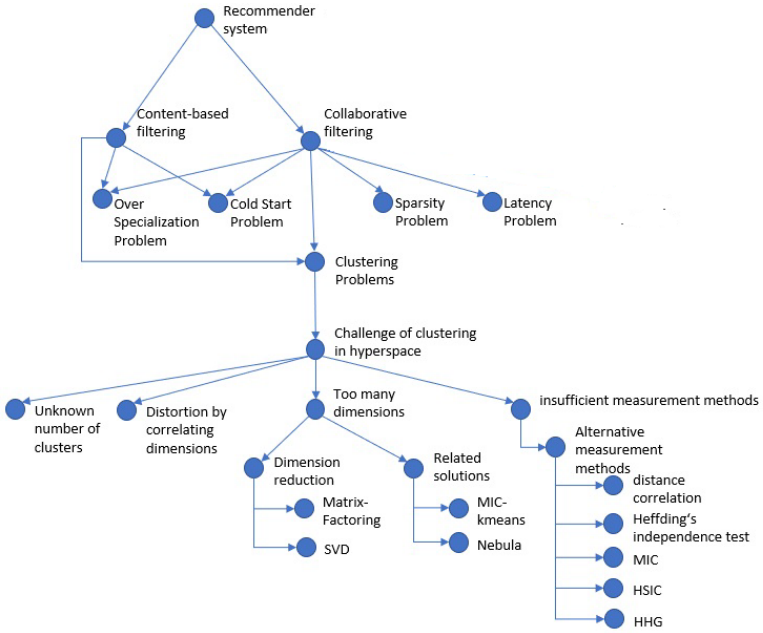


Figure 17.1: Overview of problems and solutions of recommendation and clustering in hyperspace as a tree.

17.3 Binarized Clustering

In Chapter 9 we have introduced the *bitvector machine*, a simple but effective machine learning algorithm, where feature vectors are partitioned along the medians (in each component) and converted into bitvectors that are learned. It was shown that the method accelerates both training and classification. The effectiveness of the method was analyzed theoretically for best and worst-case scenarios. Experiments on high-dimensional synthetic and real-world data showed a performance boost compared to SVMs with RBF kernel. By tabulating kernel functions, computing medians in linear-time, and exploiting modern processor technology for advanced bitvector operations, a speed-up of more than 30 for classification and one of almost 50 for kernel evaluation compared to the popular library implementations were achieved. Especially for iso-oriented, multi-clustered problems the method had qualitative advantages over the linear classifier and achieved a high classification accuracy.

The Hamming distance has the property that its values are bounded by the length of the vectors to be compared, and allows several important code optimizations, most notably the use of look-up tables and native processor instructions for kernel evaluation. For transforming the input data from floating point data to binary, medians can be computed efficiently in linear time.

To explain the problems that arise with current clustering technology, we define some synthetic data sets based on boolean vectors $x \in \{0,1\}$, like:

$$M = \left\{ \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}; \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \right\}$$

We expect to cluster the data in this set into three groups. The first four vectors could be defined as a group because they only use the first two dimensions. The next four vectors also describe a group, because they only

use the last two dimensions. The last cluster is made up of all vectors without occurrence. Problems like this are found in the area of collaborative filtering recommenders in order to find matching customers and products. In this setup different cluster algorithms were tried to solve this problem.

Algorithm	Parameters	Cluster Result
K-Means	expectedClusterCount = 3	[1, 1, 0, 1, 2, 2, 0, 2, 0, 0, 0, 0]
Spectral clustering	expectedClusterCount = 3, randomState=0	[1, 1, 0, 1, 2, 2, 0, 2, 0, 0, 0, 0]
DBSCAN	minSamples = 3	[-1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0]
OPTICS	minSamples = 3	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
MeanShift	bandwidth = 0.666	[1, 3, 4, 1, 2, 5, 6, 2, 0, 0, 0, 0]
Affinity propagation	randomState = 0	[1, 0, 0, 1, 2, 2, 3, 2, 4, 4, 4, 4]
Ward hierarchical clustering		[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]

Table 17.1: Tabular listing of the results of the bit vector observation

The various clustering algorithms and their results of the observation are listed in Figure 17.1. As discussed above, three clusters are expected. So the expected solution would be similar to $[0,0,0,0,1,1,1,1,2,2,2,2]$. It can be seen here that none of the selected algorithms can adequately solve the problem. To ensure that this observation is not just an isolated phenomenon, we have created a setup to simulate variable cases. The setup generates random bit-vectors by different dimensions, numbers of groups and group elements.

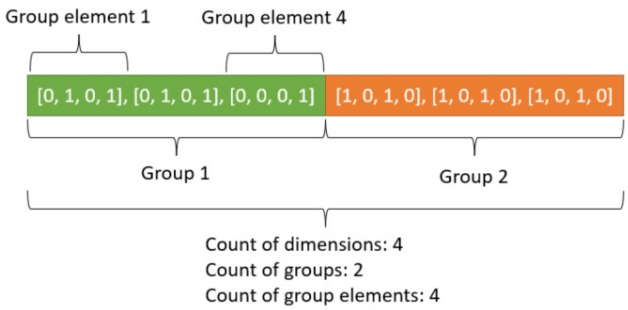


Figure 17.2: Visualization of the terms dimension, group, group elements.

Figure 17.2 shows an explanation for the terms dimensions, the numbers of groups and group elements. The example leads to two expected groups. Each expected group is associated with four rows in terms of dimensionality. Each combination of groups, group elements and dimension the clustering result was calculated 24 times. We provide the number of elements for every group with the same value.

Figure 17.3 gives an overview about the results of the simulation. We see that k-means and the spectral clustering are the best algorithms, with a hit rate of 59.56% and 56.53%. The other cluster algorithms have a hit rate under 12%.

One solution for this problem is a modified k-means clustering algorithms, that includes the dimensions as additional input.

17.4 Product Configuration

We choose the following characterization for *product configuration* tasks. A *parameter* is a property of a component and consists of a name and a range of possible value assignments. If the set of values is reduced to

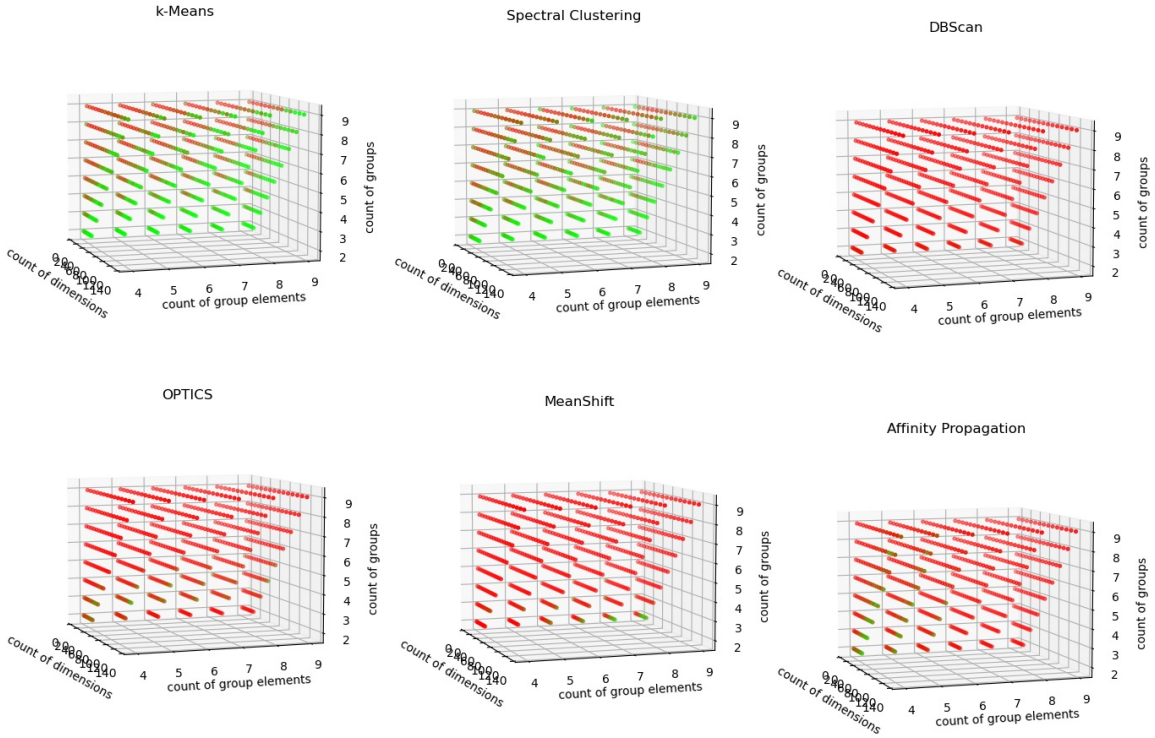


Figure 17.3: 3D overview of the results of the various clustrering algorithms.

one based on the constraints provided, or fixed by a user, it is *terminal*. The task of *product configuration* is to assemble a product from a set of components and their parameters, so that all constraints are satisfied.

An example is a car with the *components* motor, wheels, carriage, body and with the parameter *color*. In its simplest form the configuration problem can be seen as a constraint satisfaction problem (CSP). By the exponential increase in the number of configurations (in the number of parameters and product parts) and complexity of the constraints (to be propagated), manual configuration quickly becomes infeasible.

The *configuration problem* is a triple (V,D,C) of domain variables $V = \{v_1,\ldots,v_n\}$, together with their individual ranges $D = \{dom(v_1),\ldots,dom(v_n)\}$. The set of constraints C consists of two subsets $C = C_P \cup C_U$, where $C_P = \{c_1,\ldots,c_m\}$ are the rules imposed by the product and $C_U = \{c_{m+1},\ldots,c_u\}$ are the requirement imposed by the user. A *configuration* is an assignment of the variables to some value of their respective domains. It is *complete* if all variables are assigned, it is *consistent*, if all constraints are satisfied and *valid* if it is both complete and consistent.

In industrial practice, the above characterization of the configuration problem in form of a plain CSP is too restrictive, in fact there are different sources of information that must be included into the inference process. For the parts within a product family there are usually taxonomies (is-a-relations) and partonomies (has-a-relations) that are kept in ontological knowledge bases so that additional constraint propagation by calling reasoners for description logics is appropriate. To highlight the difference of ordinary and advanced constraint reasoning, we denote the process as *structured configuration*.

17.5 Recommendations for Product Configuration

We have seen that recommender systems assist users in finding items in big data sets. Recommender items can be of complex structure. For product configuration, it has been shown that recommendations increase user satisfaction. They can, e.g., be used to recommend parameters that should be evaluated next to finalize the configuration as soon as possible; explain alternatives, if the current configuration cannot be completed; recommend assignments to parameters; or complete (partial) configuration fully automatically.

Usually, recommender systems are devised to pain objects (like films or books), and less frequent to complex objects (like apparent in the configuration task). As in some cases users have problems in making their interests explicit, it is possible to first configure a product and, then, recommend possible alternatives.

Due to market interests, in industrial practice not always the strictly cheapest and fastest choice of parameters is provided to the user. The optimization criterion must be modified through computing the *utility* of the product, which then has to be minimized. Sometimes the recommendation problem induces the optimization of multiple criteria.

17.6 Algorithms

We chose two basic designs of our recommender system for product configuration, one detecting association rules, and one applying case-based reasoning.

17.6.1 Association Rule Mining

As the first approach we use a rather unusual approach for recommendation based on mining *association rules*. Given a list of transitions, the set of all possible rules of the form $A \rightarrow B$, $A, B \subseteq I$, I being the set of items, is filtered by the *minimum confidence* and *minimum support*. These rules can, in turn, be used to derive the recommendation.

For example, suppose we have the rules $\{a = 1, b = 2\} \Rightarrow \{x = 3\}$, $\{c = 1, d = 2\} \Rightarrow \{x = 5\}$, and $\{e = 3, f = 2\} \Rightarrow \{y = 1\}$. We are searching for a recommendation for x with respect to the set of decisions $\{c = 1, d = 2, g = 7, h = 3\}$ made so far. A wanted tool derivation proposes an assignment of x to 5.

In hierarchically structured data like the overlay of sequences of decisions made by the users, we have that additional problem that ancestors of any decision node is visited more often than its offsprings (Figure 17.4). Depending on the support, we get either only a very few rules or too many weak rules. This brings us to prefer *on-line* to *off-line learning*, considering the list of choices the user has made so far. We derive that some sort of balancing is needed.

Moreover, in the configuration domain there is an obvious dependence on the order of decisions. If decision a has to be made before decision c, the rule $\{b, c\} \Rightarrow a$ is not valid.

Instead of the well-known *Apriori* algorithm we take *FP-Growth* (see Algorithm 17.1) as the basis. It has a better cache reputation and adapts better to the structured configuration problem that we look at.

Both algorithms infer frequent patterns for subsequent rule induction. They rely on the *support* and the *confidence* of transactions; the first one being defined as the ratio of transactions that include all items of the rule, and the latter other one referring to the ratio of the support for the rule and the support for the trigger of the rule. To reduce the search space, the minimum support and confidence are provided as parameters.

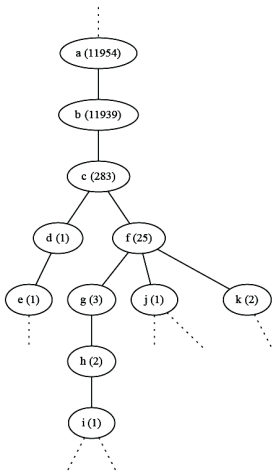


Figure 17.4: Overlay of decision lists, parameters are shown with the number of visits in brackets.

Algorithm 17.1: FP–Growth

procedure fpGrowth(*tree*, *pattern*, *minSup*)
input FP–Tree *tree*, current frequent pattern *pattern*, support threshold *minSup*
output Set of frequent patterns *patterns*
$patterns \leftarrow \emptyset$
for $node \in tree.headerTable$
  $support \leftarrow tree.getSupport(node)$
  $newPattern \leftarrow frequentPattern(pattern \cup node.item,\ support)$
  $patterns \leftarrow patterns \cup newPattern$
  $prefixPaths \leftarrow collectPrefixPaths(node)$
  $transformedPrefixPaths \leftarrow transform(prefixPaths)$
  $conditionalBase \leftarrow conditionalBase(transformedPrefixPaths)$
  $conditionalTree \leftarrow conditonalTree(conditionalBase,\ minSup).build()$
  if $\neg conditionalTree.isEmpty()$
    $patterns \leftarrow patterns \cup fpGrowth(conditionalTree,\ newPattern,\ minSup)$
return *patterns*

FP-Growth already uses a tree for its induction process: the *FP-tree* (short for *frequent pattern tree*, Algorithm 17.2), which is built in the first stage of the algorithm. The second stage of the algorithm generates frequent patterns. The main advantage of the FP-tree data structure is that the number of items is often much smaller than the number of transactions.

In both the Apriori and the FP-Growth algorithm, the derivation of rules from the frequent patterns is simple and respects the threshold for the minimum confidence required (see Algorithm 17.3).

17.6.2 Nearest Neighbor

The next approach, originally designed for classification tasks only, nowadays refers to the more general task of *case-based reasoning*. It goes back to one the simplest classification algorithm in machine learning: the task in the k-nearest neighbors algorithm, KNN for short, is to determine the k training samples that have the smallest distance to the query (according to some distance metric), and choose the one corresponding to the outcome of majority sampling.

Algorithm 17.2: FP–Tree construction

```
procedure buildTree(T, minSup)
input Transaction database T, support threshold minSup
output FP–Tree
F ← collectAllFrequentItemsWithSupportCount(T)
L ← sortInDescendingOrder(F)
headerTable ← HeaderTable()
root ← node(null)
for transaction ∈ T
    transaction ← sort(transaction, L)
    transaction ← removeInFrequent(transaction, L, minSup)
    current ← root
    for item ∈ transaction
        current ← insertNode(item, current, headerTable)
return fpTree(root, headerTable)

procedure insertNode(item, current, headerTable)
node ← null
if current.children.contains(item)
    node ← current.children.get(item)
    node.count ← node.count + 1
else
    node ← node(item, current)
    headerTable.addLink(node)
return node
```

Algorithm 17.3: Rule generation from frequent patterns.

```
procedure generateRules(l, minConf)
input frequent pattern l, confidence threshold minConf
output Set of rules R
R ← ∅; A ← generateNonEmptySubset(l)
for a ∈ 𝒜
    c ← l \ a; r ← rule(a, c)
    if confidence(r) ≥ minConf
        R ← R ∪ {r}
return R
```

When the last step of voting is neglected, classification becomes recommendation. For decision lists u and w, and a set of parameters *Assigned*, we define distance $d(u,w) = |\{x \in Assigned \mid u[x] \neq w[x]\}|$.

Cumulating this measure definition for arbitrary decision lists the to an overall function *distance* the k-nearest neighbors can be determined (see Algorithm 17.4).

Algorithm 17.4: Calculation of the k nearest neighbors.

```
procedure knn(k, o, X)
input number of neighbors to be found k, reference object o, set of objects X
output Set of k nearest neighbors of o
for x ∈ 𝒳
    compute distance(o, x)
return k elements with smallest distance
```

Algorithm 17.5: Rule generation for a specific parameter.

```
procedure generateRulesForParameter(decisionTree, parameterId)
input Decision–Tree decisionTree, support threshold minSup, conf. threshold minConf
decisionPaths ← decisionTree.prefixPaths(parameterId)
store ← decisionPathTransactionStore(decisionPaths)
patterns ← findFrequentPatterns(store, minSup)
rules ← generateRules(patterns, parameterId, minConf)
save(rules)
```

Algorithm 17.6: Calculation of the recommended value.

```
procedure recommend(productId, decisions, parameterId)
input product identification productId, previous user decisions decisions, Parameter for which the value should be recommended parameterId
output A parameter value or no match if no matching rule is available
rules ← findRulesFor(parameterId, productId)
rules ← sortByConfidence(rules)
for rule ∈ rules
   if matches(rule, decisions)
      return rule.parameterValue
return no match
```

17.6.3 Integration

In order to apply the above algorithms to the complex data recorded during configuration, a number of adaptations had to be made. For association rule learning transitions are entire decision lists and, thus, more complex, and for nearest-neighbor search we observe that the parts of the data are dynamic and represented as a tree ($k = 20$ proved to be an appropriate value). The vote is the one that maximizes the weight, i.e., for query q it is determined by

$$vote = arg\max_{v} \sum_{w \in N(k,u), w[q]=v} d(u,v).$$

In difference to the standard setting for recommendation in configuration we have a series of decisions, so that decision already made were also taken into account.

A variant of the FP-tree that we call *linked decision tree* is devised with nodes that contain information about the parameters, their value assignments, the number of decision lists it contains and a link to the next parameter. We also maintain a header table, that contains the name of the parameter and a link to the decision subtree. The core difference of a decision tree to a FP-tree is that the transactions in a decision tree are unsorted and unfiltered, and that we have a reference of a node to its parent.

For the example of parameter settings $\{a = 1, b = 2, c = 3, d = 4\}$, $\{a = 1, b = 2, c = 4, d = 4\}$, $\{a = 1, g = 2, f = 5, d = 3\}$ we have the decision tree shown in Figure 17.5. The header table labeled with the parameters contain a link to the list of nodes corresponding to that parameter. The decision tree provides fast access to the transactions. While going upwards from the set of linked nodes, common decisions can be spotted with their respecive frequencies being counted.

As a consequence, a decision tree achieves a lower compression ration compared to an FP-tree. If the tree exceeds main memory capacity, we use a graph database that is addressed by the in-memory header table.

Given the decision tree, for a given parameter rules can be derived. The tree is built once and then used as a static dictionary such that different treads can read the information concurrently, leaving room for parallelization.

The most important extension of rule induction for the configuration task is to automatically reduce the set of options by the order of decisions made so far. Moreover, rules that have no intersection with the current decision

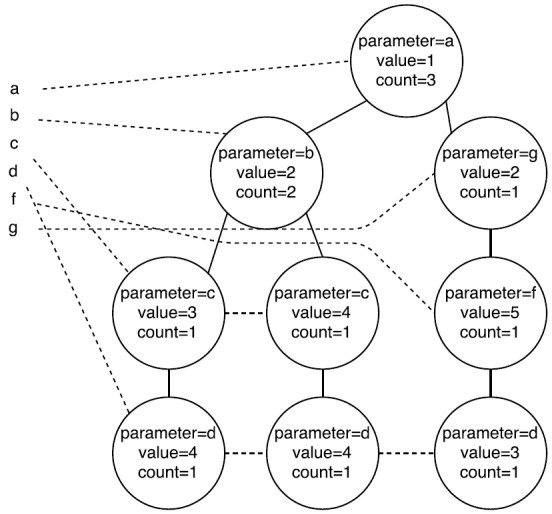


Figure 17.5: Decision tree with header table (left) and links (dashed).

Algorithm 17.7: Mapping of a parameter to a string.

procedure mapParameter(*parameter*)
input A parameter with its value and name *parameter*
output the mapped parameter as a string
return *parameter.name* + *parameter.value*

list can be ignored. Different trade-offs between the time and space for constraint propagation on the paths for filtering have been tested. At the end, we used a cache as a compromise.

The recommendation is computed in Algorithms 17.5–17.6. The software implemented in Java can be decomposed into components for importing the data, for training, a transaction gateway to export the learning results, some visualization and evaluation procedures as well as the recommendation component. For rule induction, we restricted the interface to use only singletons in their triggers. Objects stored use the JSON format. For computing distances, the configurations are serialized. We also filtered decision based on their similarity score, to avoid that two conceptually different decisions are compared.

Algorithm 17.8: Complete mapping of a parameter to a string.

procedure mapParameterName(*parameter*, *node*)
input A parameter *parameter*, current component in the tree *node*
output the mapped parameter as a string
return *concat*(*node*) + *parameter.name*

procedure concat(*node*)
if *node* = *null* **return**
else return *concat*(*node.getParent*) + *node.name* + *node.nr*

Parameter names might not be unique; think of a car with different wheels, each of which can be configured independently. As configurations are trees, we traverse the paths to a node to generate a unique identifier (see Algorithms 17.7– 17.8).

17.7 Evaluation

As data basis we took the records of a commercial configuration tool, recording customer choices during the actual configuration process. In our industrial test set we initially had configurations for 33 products, but the number of configurations was not distributed uniformly: for nine products we had less than 100 configurations, for 11 products the value was in between 100 and 1000. Two products had the maximum of 12,269 and 13,571 configurations, respectively. From this set we chose three different products:

Product	Configurations
a	3,807
b	7,828
c	8,495

We altered the size of the data basis to see scaling effects and used subsets of 100, 2,500, and 5,000 configurations each. Nonetheless, the sample set is comparably small, so that we used 10-fold cross-validation to improve the reliability of the results.

We provided three inputs to the recommender system: the product ID to denote which product is currently configured, and the complete history of decision already taken. Last, but not least the parameter had to be selected, for which a recommendation is requested.

In our experiments we looked at the following recommender system algorithm variants. KNN: weighted majority votes of the k-nearest neighbors; Simple: rule-based approach with a reference to the ordering of decision; Context: rule-based approach with a reference to the ordering *and* history of decision; and KNN Context: hybrid algorithm, an extension of strategy *Context* with machine learning based on KNNs.

In the following plots we see the outcome of an evaluation for the three different sets of product data. On the left side we have shown the prediction accuracy in terms of correctly classified instances (higher values are better), while on the right side we have denoted the percentage of wrong predictions (lower values are better). In the captions, we have added some additional information on the setting chosen. We see that the parameters of the rule data mining process have been tuned to the data set.

On the first glance, the good performance of KNN surprised: its simple learning mechanism was quite effective. However, it always answers with *yes* or *no*, while the systems based on association rules can also say *don't know*.

There is a trade-off between the two, while KNN certainly have a better accuracy in the prediction they also lead to more answers that are wrong. The rule mining algorithm, however, are less accurate but more reliable if they give a recommendation of the user's choice. This has led to the hybrid which, according to the results, offers a compromise between these two extremes.

17.8 Summary

After looking into challenges for recommender systems and issues with known recommendation approaches based on clustering binary vectors, we have designed, implemented, and evaluated a recommender system for a (commercial) configuration tool. Machine learning from recorded configurations helped to predict possible choices to be made by the user. As structured configurations are complex objects to be learned the obtained prediction rates are promising. While there is work in recommendation for configuration tasks, we provided an implementation that is applied to on real data.

From a scientific point of view, machine learning structured configurations based on training data imposes a significant challenge, as the size of the configuration space is large and the object to be learned is an entire data

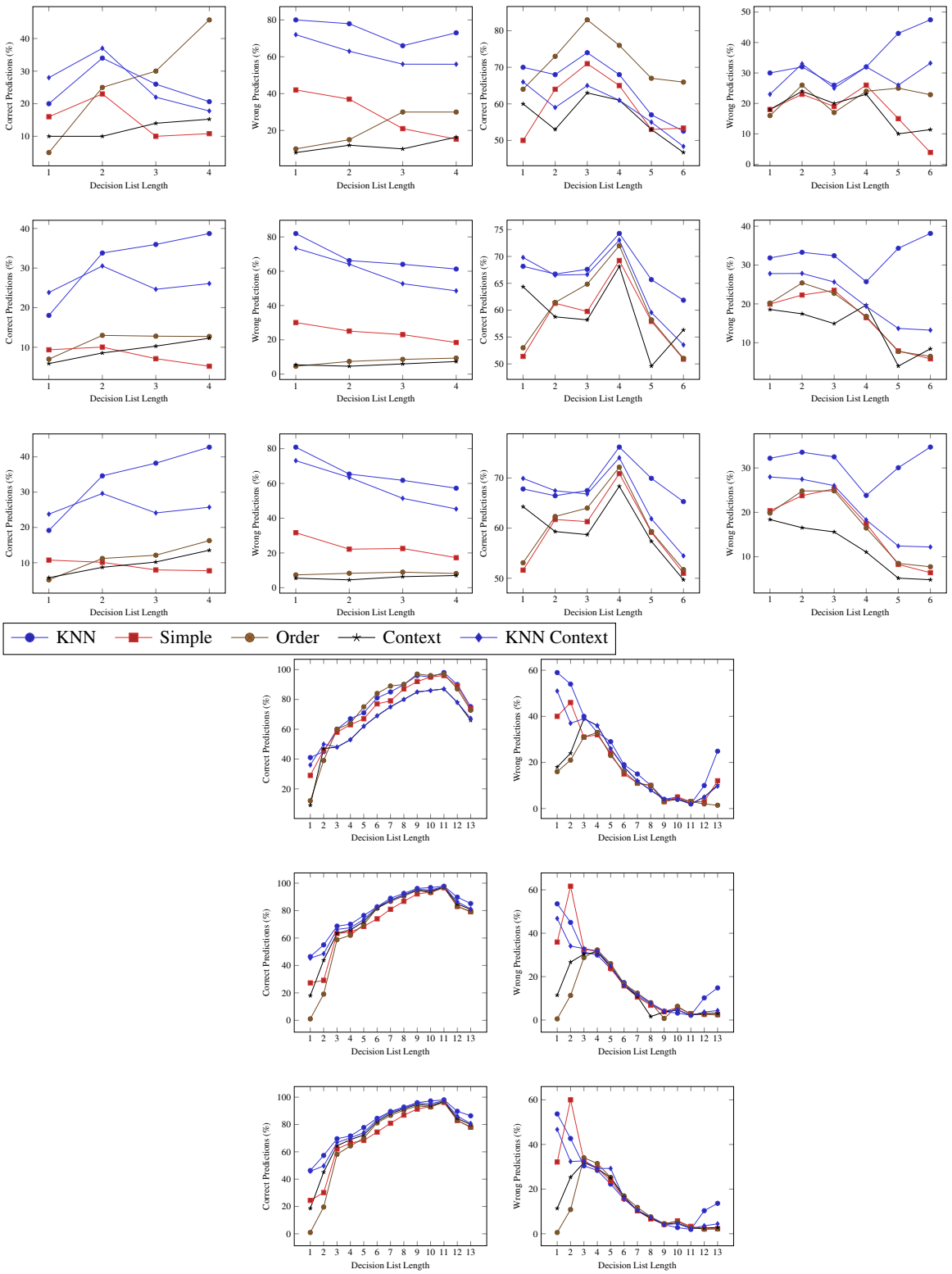


Figure 17.6: Results of evaluation: for product *a* (top left); *Simple* requests a support of 5% and a confidence of 10%, *Order* requests a support of 5% and a confidence of 30%, *Context* requests a support of 5% and a confidence of 10%, while *KNN Context* requests a support of 5% and a confidence of 50%. for product *b* (top right); *Simple* requests a support of 5% and a confidence of 12.5%, *Order* requests a support of 5% and a confidence of 25%, *Context* requests a support of 5% and a confidence of 50%, while *KNN Context* requests a support of 10% and a confidence of 30%. for product *c* (bottom); *Simple* requests a support of 5% and a confidence of 25%, *Order* requests a support of 12% and a confidence of 50%, *Context* requests a support of 15% and a confidence of 50%, while *KNN Context* requests a support of 10% and a confidence of 30%.

structure for which expressive distance metrics are non-trivial to derive. That association rules could reduce the also negative rate and KNNs reduce the false positive rate was an unexpected outcome of the experiments. It led to an ensemble of the two methods and a machine learning hybrid that suggests further investigation.

17.9 Bibliographic Notes

Mass customization [356, 624] is a current trend. The term *product configuration* is not used consistently and uniformly in the literature [423, 599, 551].

Data mining [318, 609] has be characterized as the art of making sense of data, making implicit relations explicit to the user. A basic understanding of machine learning has been given by [571]. A primer for *case-based reasoning* can be found in [624, 458], with a k-nearest neighbors approach with relation to configuration going back to [131]. The *Apriori* algorithm has been proposed by [6] and *FP-Growth* by [7, 318]

Configuration as constraint satisfaction is proposed by [261, 526]. Precursing work in recommendation for configuration tasks include [248, 336], Knowledge bases and additional control rules are devised that drive the constraint propagation and decomposition process [311, 343] and called *structured configuration* [311].

There are different types of recommender systems: the ones mentioned applying collaborative filtering, others are content-based, demographic, and knowledge-based [92, 534, 252, 251]. For the latter recommender systems, we further distinguish constraint-based [251]. and case-based [288] systems. There are many mixtures including ones that are merged using weights, switches, feature combinations or augmentation [533].

Recommender systems [533, 534, 252] usually refer to the process of *collaborative filtering* [336]. Prominent examples are the recommendation of films for users in the netflix machine learning challenge or Amazons' recommendation engine for product selection [411]. There are various approaches for recommender systems, e.g., ones by matrix factorization [411] for which the large assignment matrix is approximated by two matrices of lower dimension, whose entries are trained [614, 336],

Part IV
Applications

Chapter 18
Adversarial Planning

Effective and efficient reasoning in adversarial environments is important for many real-world applications, ranging from cybersecurity to military operations. Deliberative reasoning techniques, such as automated planning, are often restricted to static environments where only one agent can make changes through his/her actions.

While such techniques are effective and can generate non-trivial solutions, the presence of an adversary strongly influences plan quality.

18.1 Introduction

Planning[1] in static environments accounts for generating plans that are optimized, for instance, for their length, makespan or action cost. However, in environments, where an adversarial (alias competing) agent is present, such naive approach is rarely effective.

Automated Planning is an important tool for enabling deliberative reasoning of intelligent agents but many application domains consist of multiple actors – agents, independent on each other, that act in order to achieve their goals – that can either willingly or unknowingly interfere with each other. Hence, the planning approach has to be modified in order to handle multiple agents. To be more specific, in scenarios in which agents have conflicting goals such as in zero-sum games each agent has to consider a possible strategy of its opponent while generating its own strategy. Such scenarios include, for instance, competing for limited resources in games or competing for customers in on-demand transport services.

One of the best-known game-theoretic algorithms is the incremental strategy generation method called the *double oracle* (DO) algorithm. DO algorithm tackles one common problem of games: the exponential number of possibilities to choose from. The number of plans needed to achieve certain goals raises usually exponential with respect to the number of agent's actions.

DO, therefore, restricts the space of possible plans to choose from; the algorithm forms *a restricted problem* that is iteratively expanded by calculating and adding into the problem new plans as *responses* to the current strategy of the other agent from the restricted problem. Although, in the worst case, all plans have to be added into the restricted problem, it rarely happens in practice and DO algorithms are often able to find an optimal strategy using only a fraction of all possibilities.

Domain-independent tools from classical planning can also be used to model and solve a broad class of game-theoretic problems that we call *cost-adversarial planning games* (CAPGs).

[1] This chapter is based on joint work with Lukás Chrpa, Pavel Rytír, Rostislav Horcík, Jan Cuhel, Anastasiia Livochka, Álvaro Torralba, and Andrii Nyporko. It puts together and improves the work from [122, 123, 349, 124].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_18

We can define CAPGs as a two-player normal-form games specified by a planning task and a finite collection of cost functions. The first player (a planning agent) strives to solve a planning task optimally but has limited knowledge about its action costs. The second player (an adversary agent) controls the actual action costs.

Even though CAPGs need not to be zero-sum, every CAPG has an associated zero-sum game whose Nash equilibrium provides the optimal randomized strategy for the planning agent in the original CAPG. It is possible to find the Nash equilibrium of the associated zero-sum game using a cost-optimal planner via the DO algorithm. To demonstrate the expressivity of CAPGs, one can formalize a patrolling security game and several IPC domains as CAPGs.

18.2 Basics in Game Theory

We need to recall a few definitions and basic facts from game theory. A *2-player normal-form game* (shortly game) is a quadruple $G = (X, Y, u_1, u_2)$ where X (resp. Y) is a finite set of pure strategies of Player 1 (resp. Player 2), $u_1 : X \times Y \to \mathbb{R}$ (resp. $u_2 : X \times Y \to \mathbb{R}$) is a utility function of Player 1 (resp. Player 2). When the game G is played, both players choose simultaneously a strategy from their respective sets of *pure strategies* X, Y. The outcome of G for Player i is given by the *utility function* u_i. The players strive to maximize their utilities. If a game is played repeatedly, it might be reasonable for the players to randomize their strategies in order to increase their expected utilities. A *mixed strategy* for Player 1 is a probabilistic distribution $p : X \to [0,1]$. The set of all mixed strategies is Δ_X. For a mixed strategy $p \in \Delta_X$ we define its support $spt(p) = \{x \in X \mid p(x) > 0\}$. Note that a pure strategy x corresponds to a mixed strategy δ_X such that $spt(\delta_x) = \{x\}$. Set Δ_Y for Player 2 and their supports are defined analogously.

18.2.1 Nash Equilibria

The utility function can be extended to $\Delta = \Delta_X \times \Delta_Y$ by setting $u_i(p_q) = \sum_{x \in spt(p)} \sum_{y \in spt(q)} p(x)q(y)u_i(x,y)$. In other words, $u_i(p,q)$ is the expected utility of Player i if Player 1 plays p and Player 2 plays q.

An important class of games are zero-sum games, i.e., games where $u_1 + u_2 = 0$. In zero-sum games, it suffices to consider only a single utility function $u = -u_1 = u_2$ that Player 1 strives to minimize and Player 2 maximize.

A standard solution concept for games is a *Nash equilibrium*, defining stable pairs of mixed strategies. Let $G = (X, Y, u_1, u_2)$ be a game. A pair of is called a *Nash equilibrium* (NE) if for all $(p,q) \in \Delta$ we have $u_1(p,q^*) \leq u_1(p^*,q^*)$ and $u_2(p^*,q) \leq u_2(p^*,q^*)$. In other words, none of the players would change her strategy knowing the strategy of the opponent because they mutually play best responses against each other. If G is zero-sum and we have only a single utility function $u = -u_1 = u_2$, the defining condition becomes $u(p^*,q) \leq u(p^*,q^*) \leq u(p,q^*)$.

Nash proved that each game has at least one NE. In fact, a game can have more than one NE. In general games it is problematic for the players to select one among all NEs because the outcome could be different for different NEs (consider, e.g., the well-known *prisoners' dilemma*). Nevertheless, this problem does not occur in zero-sum games. The value $u(p^*,q^*)$ is always the same for any NE (p^*,q^*). Moreover, p^* is a *minimax strategy* whereas q^* is a *maximin strategy* as follows from Neumann's theorem: Let $G = (X, Y, u_1, u_2)$ be a zero-sum game and (p^*,q^*) its NE. Then,

$$u(p^*,q^*) = \min_{p \in \Delta_X} \max_{q \in \Delta_Y} u(p,q) = \max_{q \in \Delta_Y} \min_{p \in \Delta_X} u(p,q).$$

Therefore, to solve a zero-sum game, both players need to find strategies preparing them best for the worst opponent's strategy. The value $v_G = u(p^*,q^*)$ is called the *value of the game*.

A game is *almost* zero-sum if the sum $u_1(l_i,l_j) + u_2(l_i,l_j) = -c(l_i)$ does not depend on Player 2's strategy. In this kind of game, we are interested in finding the optimal strategy for Player 1, which will consist of a mixed

strategy over the possible plans to reach the goal. More formally, we call a game $G = (X, Y, u_1, u_2)$ *almost zero-sum* if $u_1(x,y) + u_2(x,y) = f(x)$ for some function $f : X \to \mathbb{R}$.

To each almost zero-sum game $G = (X, Y, u_1, u_2)$, we associate its equivalent zero-sum game $G_0 = (X, Y, u_1, u_2)$ where $u(x,y) = -u_1(x,y) = u_2(x,y) - f(x)$. An almost zero-sum game G is *best-response equivalent* to its associated zero-sum game G_0. Thus, they have the same NEs. More precisely, let $G = (X, Y, u_1, u_2)$ be an almost zero-sum game such that $u_1(x,y) + u_2(x,y) = f(x)$ and $G_0 = (X, Y, u_1, u_2)$ its associated zero-sum game where $u(x,y) = -u_1(x,y) = u_2(x,y) - f(x)$. Then, (p^*, q^*) is a NE of G if and only if it is a NE of G_0.

This is easy to validate, as $u_1(p,q^*) \leq u_1(p^*,q^*)$ iff $u(p^*.q^*) = -u_1(p^*q^*) \leq -u_1(p,q^*) = u(p,q^*)$. Similarly, we have $u_2(p^*,q) \leq u_2(p^*,q^*)$ if and only if $u(p^*,q) = u_2(p^*,q) - f(p^*) \leq u_2(p^*,q^*) - f(p^*) = u(p^*,q^*)$.

In order to solve an almost zero-sum game, it, therefore, suffices to find a NE (p^*, q^*) of its associated zero-sum game. Moreover, Player 1's utility $u_1(p^*,q^*) = -vG_0$ is just the opposite of the value of G_0 that is given by the minimax strategy $\min_{p \in \Delta_X} \max_{q \in \Delta_Y} u(p,q)$. Consequently, Player 1's utility $u_1(p^*,q^*)$ is always the same in any NE (p^*,q^*). Thus, Player 1 is indifferent on which equilibrial strategy to play. Nevertheless, the Player 2's utility may differ in different NEs because $u_2(p^*,q^*) = vG_0 + f(p^*)$, i.e., Player 1 can influence Player 2's utility.

In practical applications, we often look for an approximation of NE. We call a pair of mixed strategies (p',q') an ε-NE if both players can improve their utilities at most by when playing (p',q'). More formally, let $G = (X.Y, u_1, u_2)$ be a game and $\varepsilon > 0$. A pair $(p',q') \in \Delta$ of mixed strategies is called an ε-Nash equilibrium (ε-NE) if for all $(p,q) \in \Delta$ we have $u_1(p,q') \leq u_1(p',q') + \varepsilon$ and $u_2(p',q) \leq u_2(p',q') + \varepsilon$ The advantage of ε-NE is that it is guaranteed that there exists a NE with *small* supports.

18.2.2 Double-Oracle Algorithm

If we want to compute a NE of a zero-sum game $G = (X, Y, u_1, u_2)$ and we know its utility function u on $X \times Y$, we can find it by linear programming in polynomial time in the size of the representation of u. However, we deal with situations when both X or Y might be very large, or it might be difficult to compute the utility function for all possible pure strategies.

Algorithm 18.1: Double-oracle algorithm

Input: Zero-sum game $G = (X, Y, u_1, u_2)$, nonempty finite subsets $X_1 \subseteq X$,, $Y_1 \subseteq Y$
Output: ε-NE (p_i^*, q_i^*) of G

```
i ← 0
repeat
    i ← i + 1
    Find NE(p_i^*, q_i^*) of subgame (X_i, Y_i, u)
    Find some x_{i+1} ∈ br(q_i^*) and y_{i+1} ∈ br(p_i^*)
    X_{i+1} ← X_i ∪ {x_{i+1}} and Y_{i+1} = Y_i ∪ {y_{i+1}}
    v̄_i = u(p_i^*, y_{i+1})
    v_i = u(x_{i+1}, q_i^*)
until v̄_i − v_i ≤ ε
return (p_i^*, q_i^*)
```

To overcome these difficulties, we recall the Double-Oracle Algorithm (DO) as introduced by McMahan, Gordon, and Blu that can compute a NE (or ε-NE if it is stopped before the final iteration). It iteratively computes a NE of a subgame without evaluating the utility function in all points.

When $X' \subseteq X$ and $Y' \subseteq Y$ define the *subgame* $G' = (X', Y', u_1, u_2)$ of G by restriction of u to $X' \times Y'$. The subgames in particular iterations of DO are constructed from best responses. Given a mixed strategy $q \in \Delta_Y$,

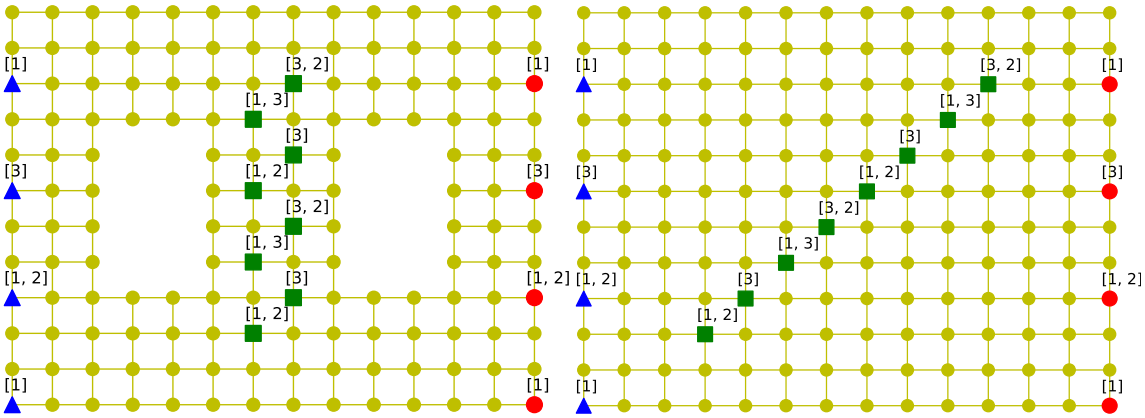


Figure 18.1: A game with 4 UAVs and 6 resources. UAVs of Player 1 are depicted by blue triangles. UAVs of Player 2 are depicted by red circles. Resources are depicted by green squares. The numbers in the brackets are sensors that a UAV has or are required for a resource to be collected.

the best response set for Player 1 is defined as $br(q) = \{x \in X \mid u(x,q) = \min_{x' \in X} u(x',q)\}$. Analogously, for $p \in \Delta_X$, the best response set for Player 2 is $br(p) = \{y \in Y \mid u(p,y) = \max_{y' \in Y} u(p,y')\}$.

The pseudocode of the DO algorithm is listed in Algorithm 18.1. The algorithm starts with the sets X_1 and Y_1 of initial pure strategies. Typically, X_1 and Y_1 are singletons. Next, these sets are iteratively enlarged by best responses and the resulting subgame is solved by an LP solver. The crucial observation regarding the convergence of DO is the fact that a best-response for Player 1 gives a lower bound on the game value v_i whereas a best response for Player 2 gives an upper bound on the game value v_i. Consequently, if these bounds are ε-close in an iteration i, the NE of the subgame (X_i, Y_i, u_1, u_2) is an ε-NE of G. Further, note that DO is not a deterministic algorithm as the best responses are not unique.

The algorithm terminates when neither of the players can add a best response strategy that improves the expected outcome from the restricted game. The NE of the restricted game matches the one in the original game, since the best response is computed over the unrestricted set of all strategies. The algorithm returns an optimal strategy but is not monotone (in the upper and lower bounds on the game value in each iteration), and might have to consider an exponential number of strategies during its computation, calling for a computational trade-off.

18.3 Incorporating Planning into the Double Oracle Algorithm

As stated before, DO considers a restricted game with a set of pure strategies for each player and where each player iteratively generates (best) response to the opponent strategy until neither player can improve its strategy. In our case, pure strategies of agents are set of plans such that each plan can be applied with a given probability. For finding the (best) response to the competitor's strategy, an agent formulates a response planning task. If it is solved optimally, i.e., the response plan has minimum cost, then the agent obtained the best response.

However, finding an optimal plan might be too expensive. Also, if heuristics are used, then suboptimal response plans are generated. If the response plan, despite being suboptimal, improves agent's strategy, the response plan is considered, and the DO algorithm continues. If none of the agents can improve its strategy, then the DO algorithm terminates.

As a first case study consider a two-player game, called *Resource Hunting*, where each player controls its fleet of unmanned aerial vehicles (UAVs) that can collect resources. The goal of each player is to collect as many resources as possible, however, each resource can be collected by at most one player. Each UAV carries one or two (different) sensors. Each resource must be collected by one or two (different) sensors. In particular, a

resource that has to be collected by two sensors requires either one UAV equipped by the corresponding two sensors, or two UAVs each equipped with the corresponding sensor.

The map of the game is modeled as a graph in which the vertices represent locations and edges connect neighboring locations. The (soft) goals for each player are to collect resources that are placed in some locations on the map. Each player controls a group of unmanned aerial vehicles (UAVs). Each UAV has at most two different sensors. Each resource requires one or two sensors for being collected.

There are two types of actions the player can take: the *move* action, which moves an UAV from one location to another such that the locations are connected by an edge, and the *collect* action, where one or more UAVs collect a resource present in the same location as the UAVs and where the UAVs possess the required sensors. The examples of two types of scenarios, the "middle" and "diagonal" ones, we use for experiments are depicted in Figure 18.1

Program 18.1: Resource-hunting PDDL domain (simplified).

```
(define (domain resource-hunting)
(:requirements :adl :fluents :durative-actions :preferences)
(:types unit resource location sensor player - object)
(:predicates (connected ?l1 ?l2 - location)
             (has-unit ?u - unit ?p - player) (at-unit ?u - unit ?l - location)
             (free ?u - unit) (at-resource ?r - resource ?l - location)
             (available ?r - resource) (sampled ?u - unit ?r - resource)
             (collected ?r - resource ?p - player)
             (can-communicate ?l - location) (has-sensor ?u - unit ?s - sensor)
             (required-sensor ?r - resource ?s - sensor)
             (required-two-sensors ?r - resource ?s1 ?s2 - sensor))
(:functions (move-cost ?l1 ?l2 - location) - number)
(:durative-action move
:parameters (?u - unit ?curpos ?nextpos - location ?p - player)
:duration (= ?duration (move-cost ?curpos ?nextpos))
:condition (and (at start (at-unit ?u ?curpos))
                (over all (connected ?curpos ?nextpos))
                (over all (has-unit ?u ?p)))
:effect (and (at end (at-unit ?u ?nextpos))))
(:durative-action sample
:parameters (?u - unit ?r - resource ?l - location ?s - sensor ?p - player)
:duration (= ?duration 10)
:condition (and (over all (at-unit ?u ?l))
                (at start (free ?u))
                (over all (at-resource ?r ?l))
                (at start (available ?r))
                (over all (has-sensor ?u ?s))
                (over all (required-sensor ?r ?s))
                (over all (has-unit ?u ?p)))
:effect (and (at start (not (free ?u)))
             (at start (not (available ?r)))
             (at end (collected ?r ?p))
             (at end (free ?u))))
[...]
)
```

The map of the scenario is modeled as an undirected graph, where vertices represent locations and edges represent connections between the locations. We define two types of actions, *move*, where an UAV moves between two connected locations, and, *collect* where one or two UAVs collect a resource if the UAV(s) have required sensors and are at the same location as the resource.

Program 18.2: Resource-hunting PDDL problem (simplified).

```
(define (problem grid)
(:domain resource-hunting)
(:objects
 p1 p2 - player
 loc-0 [...] - location
 r7 [...] - resource
 s1 [...] - sensor
 uav1 uav2 [...] - unit)

(:init
(at-resource r10 loc-63) (at-resource r11 loc-76) [...]
(at-unit uav1 loc-0) (at-unit uav2 loc-42) [...]
(available r10) (available r11) (available r12) [...]
(has-unit uav1 p1) (has-unit uav2 p1) [...]
(connected loc-0 loc-20) (connected loc-0 loc-35) [...]
(free uav1) (free uav2) (free uav3) [...]
(has-sensor uav1 s1) (has-sensor uav2 s1) [...]
(required-sensor r10 s3) [...]
(= (move-cost loc-91 loc-84) 70) (= (move-cost loc-84 loc-20) 110) [...]

(:goal ; Player1
(and (preference p1-r10-collected (collected r10 p1)) [...] ))
(:metric minimize (+ (* 100 (is-violated p1-r10-collected)) [...] )

(:goal ; Player 2
(and (preference p2-r10-collected (collected r10 p2)) [...] ))
(:metric minimize (+ (* 100 (is-violated p2-r10-collected)) [...] )))
```

Formulating (best) response planning task requires knowledge of an adversary (mixed) strategy. Computing adversary strategy by the Double Oracle algorithm is computationally expensive as a number of planning tasks has to be (optimally) solved. One can use a heuristic method that estimates when the competitor can apply its adversary actions as such information is important for setting the deadlines for agent's critical actions and thus formulating the (best) response problem.

18.4 Cost-Adversarial Planning Games

Cost-Adversarial Planning Games (CAPGs) are specified by a planning task that the first player (*P*-player) strives to solve optimally but the action costs are influenced by the second player (*C*-player). More precisely, CAPGs are 2-player normal-form games, where the *P*-player chooses a plan and the *C*-player chooses a cost function from a given collection. Even though, this interaction is simpler than other forms of multiagent adversarial planning, where both agents can select arbitrary plans, it can still capture many relevant scenarios. For example, by increasing the cost of certain actions, the *C*-player can force the *P*-player to choose alternative plans. Furthermore, this allows us to consider randomized strategies, which is very relevant in some practical applications.

Cost-adversarial planning games are almost zero-sum. Every almost zero-sum game is best-response equivalent to a zero-sum game. This allows computing the optimal mixed strategy for the *P*-player by computing a Nash equilibrium in the equivalent zero-sum game. Nash equilibrium (NE) is a standard solution concept for normal-form games. A NE of a zero-sum game can be computed in polynomial time via a transformation to a linear program. However, this approach is unsuitable for CAPGs because the resulting linear program may be too

large. Moreover, the transformation is computationally demanding. Thus, to find a NE of a CAPG, we employ the DO algorithm, which might be seen as a combination of the column and the constraint generation method used to solve large linear programs. This approach can leverage domain-independent cost-optimal planning to solve CAPGs with large state spaces efficiently.

Security games are usually modelled as Stackelberg games. A 2-player Stackelberg game is specified by the same data as a 2-player normal-form game. The difference is in what players know about their opponent's mixed strategy. In the normal-form game, the players have no knowledge. On the other hand, in Stackelberg games, one player is a leader and the other one a follower. The leader has to announce his/her mixed strategy in advance. The follower chooses his/her strategy afterwards. A solution for a Stackelberg game is a leader's mixed strategy maximizing his/her utility provided that the follower always plays his/her best response. Thus each CAPG (and in fact each 2-player normal-form game) defines also a 2-player Stackelberg game. As each CAPG is an almost zero-sum game, we can relate its solution to the solution of the corresponding Stackelberg game. More precisely, the P-player's mixed strategy from NE is the solution for the Stackelberg game provided that P-player is the leader. This follows because the P-player's equilibrial strategy is the minimax strategy. Thus he/she can announce his/her strategy publicly without providing his/her opponent with an advantage.

Suppose, for example, there is a national park attacked by poachers. For simplicity we assume that there is a single poacher, who regularly lays down a snare somewhere in the park. Locations with higher density of animals are more attractive for the poacher. On the other hand, we have a ranger who patrols in the park every day looking for the snare. However, the park is too large for the ranger to inspect each location in the park within the day. Our task is to find a probability distribution over a collection of circular paths of limited length starting and finishing at the ranger's base so that he minimizes the expected costs for not discovering the snare and the travelled distance. Formally, we model the park as a graph, whose vertices represent locations in the park and edges are the road connections between them. Each road connection has its length. One of the vertices is the ranger's base.

An example is given in Figure 18.2. The poacher's pure strategies are vertices where he can put the snare. If the ranger visits that location during his patrol, he eliminates the snare so that the poacher's utility is zero. On the other hand, if the ranger misses the location, the poacher's utility is proportional to the density of animals in that location. The ranger can execute at most k many moves during his patrol. Thus the ranger's pure strategies are paths starting and finishing in the ranger's base of length at most k. The ranger's cost is the travelled distance if he/she eliminates the snare. If he misses the snare, his cost is increased by the poacher's utility.

Consider the graph in Figure 18.2. It represents locations in the park and their road connections with their distances. The ranger is able to make at most seven moves between the locations during a patrol. If the poacher traps an animal, his utility is 10,000, and 0, otherwise. The ranger's cost is the sum of travelled distance plus the poacher's utility. We will discuss solutions to this game in two scenarios. Firstly, we assume that the poacher always traps an animal obtaining the value of 10,000 if his snare is not eliminated by the ranger. As the value 10,000 is much greater than the distances in the graph, the optimal mixed strategy for the poacher is to put the snare into the most distant locations from the base (i.e., locations 8,9). The precise probabilities on the locations 9, 8, 2, 5, 4 are respectively 0.5, 0.492, 0.003, 0.002, 0.001. On the other hand, the ranger is uniformly choosing among the paths (a) and (b) in Figure 18.3. Note that the paths consist of at most seven moves and cover altogether all the locations. The value of the associated zero-sum game (i.e., the ranger's cost) is 5,164. The poacher's utility is 5,000. Thus, the ranger saves every second animal on average.

Secondly, we consider a more realistic scenario, whose equilibrial strategies are shown in Figures 18.2 and 18.3. Suppose that the poacher, if his snare is not eliminated, traps an an-imal at 30% of cases in all locations except the location 6, where he is successful in 70% of the cases. So the poacher's utility, provided that the snare is not eliminated, in the location 6 is 7,000 and 3,000 in the remaining locations. Now the solution is not symmetric as before. The location 6 is attractive for the poacher so it pays off for him to risk putting the snare into a closer location. At the same time, the poacher should consider the locations 8 and 5 with higher probabilities 0.287 and 0.283. By this, the poacher forces the ranger occasionally to visit the left side of the graph while leaving the attractive location 6 unvisited. On the other hand, the ranger tends to visit the location 6 often to balance its attractiveness for the poacher. The value of the associated zero-sum game is 1,926. The poacher's utility is 1,633. Thus, the poacher traps an animal roughly in 16% of the cases.

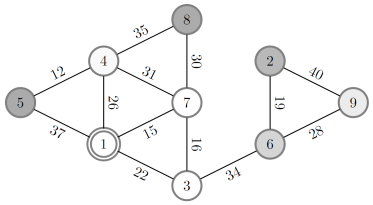
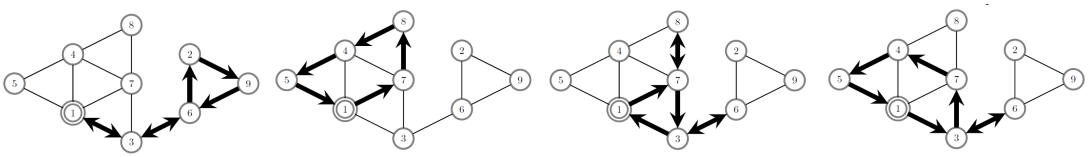


Figure 18.2: The graph representing the park. The double circled vertex denotes the ranger's base. The equilibrial strategy for the poacher is depicted with the gray nodes. The respective probabilities of the locations 8, 5, 2, 6, 9, 4 are 0.287, 0.283, 0.276 , 0.131, 0.016, 0.005.



(a) (b) (c) (d)

Figure 18.3: Equilibrial strategy for the *P*-player for the second scenario with probabilies 0.416, 0.25, 0.166, 0.166 (a)–(d).

	"Middle" Scenario					"Diagonal" Scenario				
Algorithm	Error	P1 Value	P2 Value	time(s)	Iters	Error	P1 Value	P2 Value	time (s)	Iters
Classical	0.0	3.1	2.9	3,430	34	0.0	3.11	2.89	2,590	23
PruningHeur	0.0	3.1	2.9	3,077	34	0.0	3.11	2.89	3,178	31
OrderingHeur	1.72	2.82	3.18	1,043	15	1.13	3	3	2,126	13

Table 18.1: Comparison of the approaches for scenarios with 3 UAVs and 6 resources. P1 and P2 stand for Player 1 and Player 2 respectively.

18.5 Experiments

As an optimal classical planner, we used the Fast Downward planner. with the potential heuristic or with pattern databases (ipdbs). As an optimal temporal planner we used CPT4.

18.5.1 Temporal Planning Domains

We modeled the domain in PDDL (see Figures. 18.1–18.2 for code fragments). We abstracted graphs by considering only locations of interest (e.g., with an UAV, or a resource), where length of edges between these locations correspond to minimum path length in the original graph. Reasoning with timestamps can be embedded into the model by introducing "arithmetic" and "relation" predicates that represent essential operations (e.g., adding, comparing). Enforcing ordering constraints for critical actions is done by introducing special facts representing the order of each critical actions.

Table 18.1 shows the results of the comparison. The *approximation error* is equal to the difference of the value of best response to Player 1 strategy and best response to Player 2 strategy, computed by the double-oracle algorithm. In the diagonal case, the pruning heuristics led to the highest number of iterations, i.e., the number of generated response plans until it converges, while the ordering heuristics approach led to the highest error. Whereas the latter is expectable, the former is caused by the fact that the response plans might omit collecting some resources.

domain	NE	cov	gmt	maxt	ml	tl	avgit	mit	avgP	avgC
patrol	85	90	0.7	3.3	4	1	18.2	60	5.4	7.1
transport	13	14	0.7	7.2	17	0	11.8	21	5.2	7.2
transport-road	13	14	1.2	74.1	17	0	5.3	10	2.3	3.2
data-network	3	12	0.8	33.3	16	1	14.7	21	7.7	8.3
visitall	11	16	0.5	13.7	9	0	17.5	34	9.1	13.3

Table 18.2: Overall results for an optimal planner with ipdb. NE: the number of solved tasks; cov: the number of tasks where at least one plan was produced; gmt: the geometric mean of the best-response computation times for the commonly solved tasks; maxt: the maximum best-response computation time for the commonly solved tasks; ml:the number of tasks that failed due to the memory limit; tl: the number of tasks that failed due to the time limit; avgit: the average number of iterations for the tasks solved by ipdb; mit: the maximum number of iterations for the tasks solved by ipdb; avgP: the average size of the P-player's support for the tasks solved by ipdb; avgC: the average size of the C-player's support for the tasks solved by ipdb.

18.5.2 Cost-Adversarial Planning Games

We evaluate the DO algorithm for cost-adversarial planning games in several domains. First, in the patrol domain we introduced in the previous section. Particular instances of the patrol domain were generated based on the underlying maps of locations of the 30 instances of the 2008 optimal-track version of the IPC transport domain. For each map, we considered three variants limiting the number of moves the ranger can take at most to 6, 12, and 18, respectively. Furthermore, we created variants of several domains from the IPC, namely transport, data-network and visitall. For the transport domain, we considered two variants of the C-player's pure strategies. In the first one (we call it transport), the C-player can increase by a penalty the cost of a single drive action determined by two locations x, y, and a truck t. So the penalty is applied only if the P-player applies exactly that drive action. If another truck t' drives from x to y, the P-player pays no penalty. The second variant (called transport-road) of the transport domain allows the C-player to increase simultaneously the cost of all drive actions from x to y no matter which truck applies it. The first variant clearly allows larger flexibility for the P-player to avoid the C-player's trap. In data network, the C-player can choose a server s and a scripts sc and increase simultaneously the cost of all actions processing the script sc on the server s no matter which data are processed. In visitall, we allow the C-player to choose among particular move actions where apenalty is applied. In all domains, the penalties increasing the base costs were generated randomly from the interval $[1{,}000, 10{,}000]$ for each pure strategy.

The implementation of DO can be used with any optimal planner to compute the P-player's best response. The overall results with all considered optimal planners are shown in Table 18.2. The optimal planner was able to solve the first planning task in the initialization of DO, which corresponds to the original cost function disregarding the C-player. This translates into finding also the NE in most cases. Interestingly, the difference between NE and cov suggests that inmost cases, if the underlying planner is able to find an optimal plan, the DO algorithm will terminate with a satisfactory NE, too. Furthermore, Table 18.2 shows the average, maximum numbers of iterations and the average size of the P-player's and the C-player's support.

18.6 Summary

Planning in zero-sum games concerns of finding plans that maximise the reward (or minimise the cost) for accomplished (or failed) goals in the presence of an opponent sharing the same goals such that only one of the actors can accomplish a given goal. Automated Planning can be incorporated into the DO algorithm.

Planning in adversarial environments (e.g., zero-sum games) requires to predict the strategy of the competitor, so the agent can optimize its plan accordingly.

To generate optimal mixed strategies (in Nash Equilibrium), DO can take tens of iterations until it converges (and none of the player can improve its strategy) even for smaller tasks. It involves to cost-optimally solve two planning tasks in each iteration, which is computationally expensive.

We introduced cost-adversarial planning games and showed which problems can be modelled by these games. In particular, we illustrated how to model patrolling security games within this formalism. Further, we showed how to solvethem using DO together with the tools from optimal classical planning and experimentally evaluated our solution method.

It turns out that the runtime of optimal planners can be substantially influenced by the chosen cost function. This suggests that CAPGs might serve as a reasonable benchmark to test optimal planners with respect to several cost functions.

18.7 Bibliographic Notes

Actions of other agents can be represented as exogenous events [152]. There is a range of techniques that tackle plan generation and execution under presence of exogenous events. For example, there are techniques based on Markov Decision Processes (MDPs) [409], Monte-Carlo Tree Search [504], or reasoning about "dangerous states" [121]. They do not consider scenarios where agents have conflicting goals and, hence, hinder each other their pursuit towards goals.

To explicitly reason about adversaries while generating plans, *game-theoretic* methods have to be leveraged. Importantly, agents may need to randomize over several plans, so the other agents have uncertainty about which plan is going to be executed making it difficult for them to exploit such a strategy [440]. Existing techniques involving planning and game theory focus on *congestion games* where the task is to find an optimal robust multiagent plan for non-cooperating agents [375] or on *Stackelberg games* where the task is to find a pure plan of the leader that is robust against the adversary [587]. Such techniques are not able to find randomized strategies.

While there are several successful applications of game-theoretic algorithms in practice, for example in domains of physical security [608] or protecting wildlife [250], most of the methods used for scaling-up are domain-dependent and their transferability to other domains is limited.

The concept of planning in adversarial environment is not new [21]. Succinct symbolic representations of state sets helped generating optimistic and strong cyclic adversarial plans [366, 127], a setting conceptually related to FOND planning [400]. Such a setting, however, has to explore most if not all alternatives (in analogy to traditional game-tree methods such as minimax). Monte-Carlo Tree Search (MCTS) and Online Evolutionary Planning have been applied in adversarial environments such as the Hero Academy game [377], or Starcraft [378]. Deep Reinforcement Learning (DRL) has shown impressive results in Starcraft [634] and other (adversarial) domains such as the games of Chess or Go [577]. MCTS and DRL approaches work "online": they select the most promising action (or move) in the current state of the environment, and they continue to do so until the terminal state is reached.

The idea of combining planning and game theory has appeared in previous works, although mostly with different goals. Often, the goal was to update the planning formalism to handle multiple agents and multiple goals the agents can pursue [77, 79]. Delete relaxation has a tradition in classical planning [342, 62] as well as in temporal planning [134].

A body of work concerning non-cooperative multiagent planning exploits game theory for generating plans for each agent while minimizing conflicts with plans of other agents. Resolving such conflicts can be done by translating the task into an invertible planning problem [274], or by selecting the best plan for each agent from a set of pre-computed plans using a two-game approach [374]. Closer to the above work, the conflicts can also be resolved by a *best-response* approach that iteratively improves plans of each agent [373]. Such an approach has been used for planning Electric Autonomous Vehicles [375]. These works, however, focus on *congestion*

games, for which a single plan can be optimally robust (a pure equilibrium is guaranteed to exist for this class of games). This, however, is not true for most of the non-cooperative games and adversarial scenarios. [587] used game theoretic framework of Stackelberg games and seek a pure plan of the leader that is robust against actions of the adversary. Again, we seek a possibly randomized strategy which poses computation challenges that are not present when restricting to pure strategies.

There are several existing methods that use the double-oracle incremental strategy generation method. The original work by [471] was used in the setting where one player sought an optimal way to get through an area unobserved while the other player placed the surveillance cameras. In that work and many other follow-up works (e.g., see [364, 75]), the standard assumption is that the best response algorithm is capable of computing the optimal plan (or at least a best response with a bounded error) given the strategy of the opponent. On the other hand, the recent work combined reinforcement learning with double-oracle algorithm [436, 643] on domains where computing (approximate) best response is not possible.

In terms of using domain-independent planning algorithms for computing the (best) response plans, finding an optimal plan that accounts for best response is often computationally harder than finding any plan [331]. Many planners produce sub-optimal plans and improve them until the allocated time expires or the plans cannot be further improved [627]. [550] studied how effective and efficient is to combine DO with domain-independent planning algorithms while considering varying time limits for providing the response plans and varying granularity of the underlying planning tasks.

From the planning side, [587] used the game-theoretic framework of *Stackelberg games* for generating robust plans against actions of the adversary. In a similar spirit, *Plan Interdiction Games* have been proposed to describe the problem of attackers and defenders, where the former plans to intrude a computer network, while the latter tries to prohibit attackers' actions [445, 636]. A recent work about "Counterplanning" goes in a similar direction as one agent tries to invalidate landmarks required by the opposite agent [523]. Planning-based techniques work offline, i.e., they generate plans upfront, which are then executed (as they are).

Chapter 19
Model Checking

Model checking is the automated process of checking the correctness of one piece of software with another one. Due to uncertainty of its execution order or of its inputs, the system under consideration shows non-deterministic behavior. As the general problem of checking correctness is undecidable, model checking often resorts to validating compliance with respect to temporal specification. Given a model M and some property specification ϕ the task is to determine if the model formally implies ϕ, written as $M \models \phi$. The simplest specifications are safety properties and include global invariances and variable assertions. In automata-based model checking, the property specification is provided in some form of temporal logics, which is compiled to an infinite-state automata that runs concurrent to the model execution. Model and specification together result in a larger state space, which has to be analyzed for the absence of error traces. Model checking usually amounts to exploring the corresponding state space graph.

In this chapter we accelerate state space exploration for explicit-state model checking by executing complex operations on the graphics processing unit (GPU). In contrast to existing approaches to enhance model checking performing parallel matrix operations on the GPU, we parallelize the breadth-first layered construction of the state space graph. To prevent revisiting of already explored states, all processed states are stored. If a state is generated, it is first checked against the set of stored states. Due to the huge number of states and their large sizes, time and memory demands for analyzing systems rise rapidly. For model checking safety properties, a complete scan of the reachable (possibly reduced) search space suffices.

For efficient processing on the GPU, the input model is translated to the reverse Polish notation, resulting in a representation as an integer vector. The GPU exploration algorithm then divides into two parallel stages. In the first stage, each state is replaced with a Boolean vector to denote which transitions are enabled. In the second stage, pairs consisting of replicated states and enabled transition IDs are copied to the GPU, then all transitions are applied in parallel to produce the successors. Bitstate hashing is used as a Bloom filter to remove duplicates from the set of successors in RAM.

19.1 Introduction

In[1] the last few years there has been a remarkable increase in performance and capabilities of graphics processing units (GPUs). Whereas quad-core CPU processors have become commonplace, in the years to come core numbers are likely to follow Moore's law. This trend to many-core processors is already realized in graphical processing units. Modern GPUs are not only powerful, but programmable processors featuring high arithmetic capabilities and memory bandwidths. Moreover, high-level programming interfaces have been designed for

[1] This chapter is based on joint work with Damian Sulewski, Dragan Bosnacki, Anton Wijs, and Peter Kissmann. It puts together and improves the work work from [67, 68, 603, 226].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_19

using GPUs as ordinary computing devices. Current GPUs, for example, feature up to thousands of scalar processing units per chip.

The highly parallel GPU has rapidly gained maturity as a powerful engine for computationally demanding numerical operations. The access of it is streamed, using a kernel function given to every scalar processing unit. We apply general purpose GPU (GPGPU) technology to the state space generation for explicit-state model checking. During the construction of the state space, we detect and eliminate duplicates and check a visited state for possible violation of so-called safety properties. We apply breadth-first search (BFS) and can return counterexamples of minimal length. It includes checking enabledness and generating the successors on the GPU. FAM for the exploration is mainly limited by the size of the Bloom filter. Storing full state information for expanding a state in RAM is optional, as the search frontier is managed on external memory. Eventually, the entire reachable state space has been flushed to disk.

The state space generation algorithm is divided into two stages, executed on the GPU: 1) Checking enabledness, i.e., testing the applicability of transitions against the current state; 2) Generating the set of successors (one for each enabled transition and explored state). The core reason to operate in two subsequent stages is to maximize space utilization of the graphics card. To please the GPU's computational model, the reverse Polish notation is chosen for achieving a flat bracket-free representation of expressions, since it offers the possibility to concatenate all transition descriptions to one integer vector, yielding a memory- and time-efficient exploration.

After generating the successors, they must be checked for duplicates against the list of expanded states. This can be done with either a complete method or with an incomplete but usually faster hashing method. We were able to exploit multiple threads running on the CPU for parallelizing the access to the hash table. We preferred partial search methods, because otherwise, for multi-threaded memorization at high-end exploration performance, a non-trivial lock-free hash table implementation would be needed.

The GPU model checker (CuDMoC) takes the same input format as DiVinE, namely, DVE, but shares no code. By changing the parser, however, the algorithms can be integrated to any other explicit-state model checkers, including Spin. We assume Cuda supporting NVidia hardware, but there are trends on GPGPU programming with other vendors, too.

For each of the two exploration stages, we obtain significant speed-ups of more than one order of magnitude for analyzing benchmark protocols on the GPU. In BFS, hashing contributes only a small fraction to the overall performance, so that we compute the hash values on the CPU.

19.2 GPU Essentials

Some of the design decisions in GPU model checking are closely related to the architecture. Thus, insights into GPU architecture are essential. GPUs have multiple cores, but the programming and computational models are different from the ones for multi-core CPUs. GPU programming requires a special compiler, which translates the code to native GPU instructions. Roughly speaking, the GPU architecture is that of a vector computer with the same function running on all processors. GPU architectures have different layers for accessing memory. Moreover, nowadays GPUs forbid common writes to a memory cell but support a limited form of concurrent read.

The numbers of cores on the GPU clearly exceed the ones on the CPU, but GPUs are limited to stream processing. While cores on a multi-core processor work autonomously, the operations of cores on the GPU are strongly correlated. One representative is characterized in Figure 19.1. With SLI, Tesla or Fermi technologies, more cores and larger amounts of memory are available.

A scalar *core* is a streaming processor (SP), capable of performing single precision arithmetic. SPs are grouped together with a cache structure and two special function units (performing, e.g., double precision arithmetic) to a streaming multiprocessor (SM). Texture processing clusters (TPCs) form the top-level architecture and combine

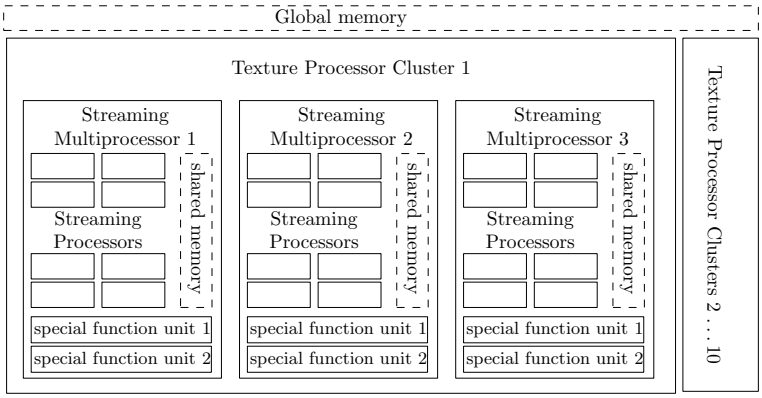


Figure 19.1: Prototypical GPU architecture.

SMs with a second cache. Since the cores are similar to an SIMD technology and operate on a lower frequency than the CPU, a linear speed-up is not to be expected.

Memory is structured hierarchically, starting with the global memory (video RAM, or VRAM). Access to this memory is relatively slow, but can be accelerated through *coalescing*, where adjacent accesses are combined to one. Each SM includes some KB of memory (shared RAM or SRAM), shared between all its SPs and accessed at a speed compatible to a register. Additional registers are also in each SP. Data must be copied to VRAM to be accessible.

The *kernel* function executed in parallel on the GPU is driven by *threads* that are grouped together in *blocks*. The TPC distributes the blocks on its streaming multiprocessors in such a way that none of the SMs runs more than a thousand threads. A block is not distributed among different SMs. This way, for a given maximal block size, at most a few blocks can be executed by one SM. Each TPC schedules several parallel threads, providing the same chunk of code to all its SMs. Since all the SPs get the same chunk of code, SPs in an *else*-branch wait for the SPs in the *if*-branch, being idle. After the threads have executed a chunk the next chunk is executed. Blocks are executed sequentially on all the resources available. Threads which are waiting for data can be parked by the TPC, while the SPs work on threads which have already received the data.

19.3 Probabilistic Model Checking

Algorithms for parallel *probabilistic model checking* on general purpose graphic processing units target the numerical components of the traditional sequential algorithms. In particular, they capitalize on the fact that in most of them operations like matrix–vector multiplication and solving systems of linear equations are the main complexity bottlenecks.

Since linear algebraic operations can be implemented very efficiently on GPUs, parallel algorithms show considerable runtime improvements compared to their counterparts on standard architectures. We implemented such parallel algorithms on top of the probabilistic model checker PRISM. The prototype implementation was evaluated on several case studies, in which significant speedups over the standard CPU implementation of the tool were observed. The work mainly considered discrete-time Markov chains (DTMCs) and the probabilistic computation tree logic (PCTL). It shows that matrix-vector multiplication and solving systems of linear equations are the corner-stones of most of the algorithms for probabilistic model checking.

Two algorithms that are parallel adaptations of the iteration method of Jacobi for solving linear equations were presented. Jacobi was chosen over other methods that usually outperform it on sequential platforms because of its lower memory requirements and potential to be parallelized because of fewer data dependencies. The

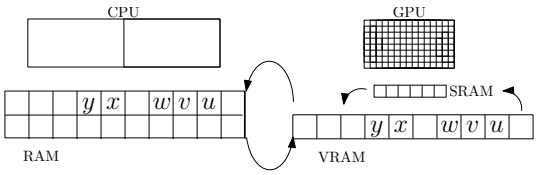


Figure 19.2: GPU-based model checking with different sorts of memory and processing cores.

algorithms feature sparse matrix–vector multiplication. It required a minimal number of copy operations from RAM to GPU and back.

19.4 GPU Breadth-First Search

In the following, we provide the essentials for breadth-first explicit-state model checking on the GPU. We show how to test enabledness for a set of states in parallel, and – given all sets of applicable transitions – how to generate the successor state sets accordingly. We restrict ourselves to BFS for generating the entire search space, since it is sufficient for verifying the safety properties. Even for model checking full LTL, which we do not address, efficient BFS state space generation is often a crucial step.

We assume a hierarchical memory structure of SRAM (small, but fast parallel access) and VRAM (large, but slow parallel access) located on the GPU, together with RAM located on the motherboard. The setting, illustrated in Figure 19.2, indicates the partition of memory into cells and of processing units into cores.

We observed that duplicate detection and elimination is not as CPU-inefficient in practice as we have expected. This is maybe due to the large number of successors that are already eliminated within one BFS layer. In the instances we looked at transition enabledness checking and successor generation were identified as the main performance bottlenecks. As the BFS search frontier is also stored on disk, we can save RAM by reading states to be expanded in blocks.

The intuition is to dispatch a set of operations to the GPU. For each BFS layer, the state space enumeration is divided into two main computational stages that are called in Algorithm 19.1; Stage 1: generate sets of enabled transitions based on checking the transition guards in parallel; and Stage 2: generate sets of successors based on applying transition effects in parallel. The codes for checking enabledness (Algorithm 19.3) and generating the successors (Algorithm 19.4) reflect that each processing core selects his share based on his group and thread ID. For duplicate detection, a Bloom filter is provided.

In the first stage, a set of enabled transitions is generated by copying the states to the VRAM and replacing them by a bitvector. In the second stage, sets of all possible successors are generated. For each enabled transition a pair, joining the transition ID and the explored state, is copied to the VRAM. Each state is replicated by the number of successors it generates in order to prevent memory from being allocated dynamically. The third stage removes all duplicates, e.g., by bitstate hashing. For a better compression in RAM, we separate the search frontier from the set of visited states on disk.

19.4.1 Preparing the Model

To check the transitions enabledness, a representation of them must be accessible by the GPU cores. While an object-oriented data structure – where each expression in a process is realized as an object linked to its substructures – might be a preferable representation of the model for CPU access, such a representation would be less effective for GPU access.

Algorithm 19.1: Breadth-first search on the GPU.

Input: initial state: s_0
Output: set of all reached states: *Closed*

Open $\leftarrow \{s_0\}$;*Closed* $\leftarrow \emptyset$; *Enabled* $\leftarrow \emptyset$; *Successors* $\leftarrow \emptyset$; ;; initialize search
while (*Open* $\neq \emptyset$) ;; repeat until search terminates
Stage 1 - Generate sets of enabled transitions
  while ($|Enabled| \neq |Open|$) ;; until all frontier states are processed
    $VRAM \leftarrow \{u \in Open \mid VRAM$ **not full**$\}$;; Copy nodes to VRAM
    $Enabled \leftarrow Enabled \cup GPU\text{-}MarkEnabled(VRAM)$;; GPU function
Stage 2 - Generate sets of successors
  while (*Enabled* $\neq \emptyset$) ;; Until all transitions processed
    $VRAM \leftarrow \{(t,s) \mid t \in Enabled$ and *s fits* $t \in Open \wedge VRAM$ **not full**$\}$
;; Move state copies for all enabled transitions of a state to VRAM
    $Enabled \leftarrow Enabled \setminus \{t\}$;; remove transitions from *Enabled*
    $Successors \leftarrow Successors \cup GPU\text{-}GenerateSuccessors(VRAM)$;; GPU function
  $Open \leftarrow \emptyset$; ;; prepare next layer
  $Successors \leftarrow Successors \cap Closed$; ;; remove explored states from successors set
  $Closed \leftarrow Closed \cup Successors$; ;; extend set of explored states
  $Open \leftarrow Successors$; ;; add new layer to the search frontier

As described in Section 19.2, the GPU's memory manager prefers sequential access to the data structures. Moreover, to use coalescing many threads must access the same memory area in parallel. Hence, to speed up the access to the model, data should reside in the SRAM of each multi-processor. This way a fast randomized access can be granted, while the available space shrinks to at most SRAM size. Another positive aspect of storing the model directly in the SRAM arises from the fast accessibility by all threads of a multi-processor, so that the model must be copied only once from the VRAM to RAM.

Since the GPU should not access RAM and pointer manipulation on the GPU is limited, it is necessary to rewrite the transition guard labels to be evaluated. This description has to be efficient in terms of memory and evaluation time, since the size of the VRAM is small (compared to the computational power of the GPU). Furthermore, all transitions should be moved into one memory block to take advantage of fast block transfers on the express bus.

19.4.1.1 Parsing the DVE Language

To use the benchmark protocols provided by the BEEM library DVE was chosen. The underlying theoretical model of the DVE language is that of communicating finite state machines and consists of several parts, structured hierarchically, and identified as global variables and several processes on the top level. Processes are divided into local variables, states and transitions, while transitions consist of guards and effects represented by Boolean formula and variable assignments, respectively. Transitions are assigned to states and indicate which state to activate if the transition is enabled. Given the process is in state *s*, only transitions assigned to *s* should be checked. If a guard evaluates to true, the transition is enabled and the effects should be applied, assigning the process a new state and optionally new values to some global or local variables. The example in Figure 19.3 shows the Anderson protocol with only one process. The array *Slot* and the variable *next* are global, while *my_place* is a local variable. The process can be in one of five states named as *NCS, p1, p2, p3, CS* where *NCS* is the initial state. Note that transitions 2 and 3 cannot be applied concurrently, since only one of the guards can be true.

Based on knowing the grammar, the model description can be parsed, and a syntax tree constructed. To store different variable assignments and indicate in which state a process currently is, a byte vector can be used. Figure 19.4 describes the state vector assigned to the example. Necessary space for each global variable is reserved, followed by a byte indicating the current state of each process, and combined with space for the local variables for each process.

```
byte Slot[1] = {1 };
byte next=0;
process P_0 {
 byte my_place;
 state NCS, p1, p2, p3, CS;
 init NCS;
 trans
 NCS -> p1 { effect my_place = next, next = next+1; },
 p1 -> p2 { guard my_place == 3-1; effect next = next-3; },
 p1 -> p2 { guard my_place != 3-1; effect my_place = my_place%3; },
 p2 -> p3 { guard Slot[my_place] == 1;  },
 p3 -> CS { effect Slot[my_place]=0; },
 CS -> NCS { effect Slot[(my_place+1)%3]=1;};
}
```

Figure 19.3: Fragments of the Anderson (1) protocol in DVE input language.

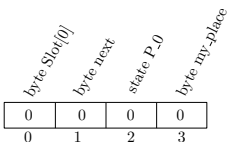


Figure 19.4: State vector representing the example.

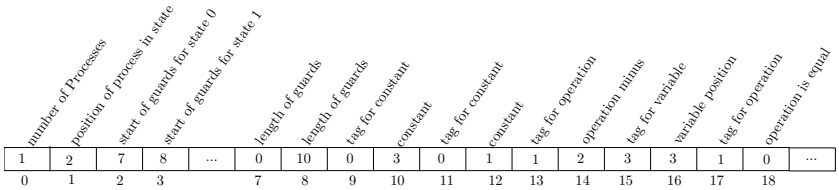


Figure 19.5: Fragments of the transition vector.

The challenge is to store the representation of the transitions efficiently. On the GPU, the *reverse Polish notation* (RPN), i.e., a postfix representation of Boolean and arithmetic expressions, is very effective. It is used to represent all guards and effects of the model in one integer array. This array is partitioned into two parts, one for the guards, the other for the effects. A prefix assigns the guards and effects to its processes after creating the array. In addition to the guards, each transition indicates the goal state the process will reach after applying the effects. *Tokens* are used to distinguish different elements of the Boolean formulas. Each entry consists of a pair (token,value) identifying the action to take. Consider the guard starting at position 8 of the array presented in Figure 19.5 representing the guard of the second transition in the example (`my_place==3-1;`). It is translated to the RPN as an entry of length 10 using tokens for constants, arithmetic operation and variables. Constant tokens, also defining the type of the constant, are followed by the value. Arithmetic tokens identify the following byte as an operator. One special token is the variable token, there is no distinction in arrays or variables, since variables are arrays of length 1, so the token defines the type of the variable and is followed by the index to access it in the state.

This yields a pointer-free, compact and flat representation of the transition guards. Converting the protocol to the RPN notation and copying it to the GPU is executed before the model checking process starts. Using this representation a check for enabledness of transitions in a process boils down to three steps: 1) checking the state the process is in, by reading its byte in the state vector; 2) checking which transitions to check by reading the global prefix of the integer vector describing the model; evaluation of all guards dependent on the actual state and process on a stack.

To enable the transition given its ID, the representation starting at the position given by the second array has to be evaluated. The advantage of this approach is being able to copy all information needed for the model checking process into two blocks. Given that all guards and effects, respectively, are in adjacent memory cells, we have a stream access for evaluating a large number of guards.

Algorithm 19.2: *CheckGuard*: checking a guard for a state on the GPU.

```
Global: expression vector guards
Input: state vector state ; transition trans
Output: true, if guard evaluation was successful; false, otherwise

pos ← start_of_guards(trans);
while pos < start_of_guards(trans) + length_of_guards(trans);
  if is_constant(guards[pos]) push guards[pos+1] on top of stack;
  if is_operator(guards[pos]) pop var1 and var2 from stack;
    result ← var1 guards[pos] var2;
    push result on top of stack;
  if is_variable(guards[pos]) push state[guards[pos+1]] on top of stack;
  pos ← next_guard(trans, pos);
return result;
```

19.4.2 State Exploration on the GPU

For accelerating the joint exploration of states, we executed both the enabledness check and the generation of successors on the GPU, parallelizing (the essentials of) the entire model checking process. We exploit the fact that the order of explorations in one BFS-layer does not matter, so that no communication between the threads nor explicit load balancing is required. Each processor is simply assigned to its share and starts operating. Duplicate detection is delayed on the GPU and delegated to the CPU.

Still, there are remaining obstacles in implementing a fully-fledged model checker on the GPU. First, the state size may vary during the verification. Fortunately, common model checkers provide upper bounds on the state vector size or induce the maximal size of the state vector once the initial state has been read. Another technical challenge is that the GPU kernel (though being C-like) does not exactly match the sources of the model checker, such that all methods being called have to be ported to Cuda.

19.4.2.1 Checking Enabledness on the GPU

In the state exploration routine first, all transitions are checked, and then, the enabled ones are fired. Before the execution the transition vector is copied to the SRAM for faster access. All threads access in parallel the VRAM and read the state vector into their registers using coalescing. Then all threads access *transitions*[0] to find the number of processes in the model. Next, all threads access *guards*[1] to find the state the first process is in. At this point in time, the memory access diverges for the first time. Since processes have reached different states at different positions in the search space, different guards must be evaluated. This does no harm, since the transition vector is accessible in the SRAM and all access is streamed. After collecting the necessary information, all threads call Algorithm 19.2. A stack consisting of pair entries (token,value) is used to evaluate the Boolean formulas. The checking process boils down to storing the values on the stack, and executing all operations on the two entries on top of the stack. The stack is used as a cache for all arithmetic operations and given that an assignment is found, the value on top of the stack is written to the state.

In the first stage the VRAM is filled with states from the *Open* list. Then, Algorithm 19.3, executed on the GPU, computes a bitvector B of transitions, with bit B_t denoting whether or not transition t applies. The entire array, whose size is equal to the number of enabled transitions, is initialized to *false*. A bit is set if a transition is enabled. Each thread reads one single state at a unique position defined by its ID and computes the set of its enabled transitions. For improved VRAM efficiency we allow the vector of transitions to overwrite the states they are applied to. Therefore, we utilize the fact that the number of transitions in a protocol is a constant that does not exceed the size of the bitvector representation of a state. After having checked all transitions for enabledness, the bitvectors are copied back to RAM.

Algorithm 19.3: *GPU-MarkEnabled*: GPU kernel for transition enabledness check.

Global: transition *guards* in reverse Polish notation
Input: state vectors $\{s_1,\ldots,s_k\}$ to check for enabledness
Output: array of transition sets $\{t_1,\ldots,t_k\}$ (overwrites state vectors with bitvectors)

```
for each group g do                                          ;; partially distributed computation
  for each thread p do in parallel                           ;; distributed computation
    B ← (false,...,false)                                    ;; clear enabledness bitvector
    for each possible transition t for s_{g·sizeof(g)+p} do  ;; select state transitions
      B[t] ← CheckGuard(s,t)                                 ;; check enabledness and set according bit
    s_{g·sizeof(g)+p} ← B                                    ;; overwrite selected state
return {s_1,...,s_k}                                         ;; return overwritten states to CPU
```

Algorithm 19.4: *GPU-GenerateSuccessors*: GPU kernel for successor generation.

Global: transition postconditions effects in reverse Polish notation *effects*
Input: set of pairs (transition,state) $\{\{t_1,s_1\},\ldots\{t_k,s_k\}\}$
Output: set of successors (explored nodes are overwritten)

```
for each group g do                                          ;; Partially distributed computation
  for each thread p do in parallel                           ;; Distributed computation
    s_{g·sizeof(g)+p} ← Explore(effects, t_{g·sizeof(g)+p}, s_{g·p})   ;; Generate successor
return {s_1,...,s_k}                                         ;; Feedback result to CPU
```

To evaluate a postfix representation of a guard, one scan through its representation suffices. The maximal length of a guard times the number of groups thus determines the parallel running time, as for all threads in a group, the check for enabledness is executed concurrently.

19.4.2.2 Generating the Successors on the GPU

After having fixed the set of applicable transitions for each state, generating the successors on the GPU is relatively simple. First, we replicate each state to be explored by the number of enabled transitions on the CPU. Moreover, we attach the ID of the transition that is enabled together with each state. Then, we move the array of states to the GPU and generate the successors in parallel.

For the application of a transition to a given state, similar to processing the guards, the effect expressions have been rewritten in reverse Polish notation (see Algorithm 19.4). Since this static representation resides in VRAM for the entire checking process, and since it is addressed by all instances of the same kernel function, its access is fast. The cause is that broadcasting is an integral operation on most graphics cards.

Each state to be explored is overwritten with the result of applying the attached transition, which often results in small changes to the state vector. Finally, all states are copied back to RAM. The run-time is determined by the maximal length of an effect times the number of groups, as for all threads in a group we generate the successors in parallel.

19.4.2.3 Duplicate Checking on (Multiple Cores of) the CPU

Since the successors are generated in parallel, an efficient parallel method is necessary to detect duplicates by checking the current state against the list of explored nodes. As in the Spin model checker, we use double-bitstate hashing as a default. Looking at the number of states explored, the error probability for tens of gigabytes

Table 19.1: Cross-comparing different versions of the GPU model checker. Running times given in seconds.

	CuDMoC			
Protocol	1 Core CPU	1 Core + GPU	8 Core + GPU	States
anderson (6)	235	25	20	18,206,914
anderson (8)	1381	669	440	538,493,685
at (5)	404	36	29	31,999,395
at (6)	836	170	119	160,588,070
bakery (7)	296	30	28	29,047,452
bakery (8)	3603	250	182	253,111,016
elevator (2)	334	30	23	11,428,766
fisher (3)	41	10	9	2,896,705
fisher (4)	22	7	7	1,272,254
fisher (5)	1692	126	86	101,027,986
fisher (6)	107	16	13	8,321,728
fisher (7)	4965	555	360	386,281,613
frogs (4)	153	20	17	17,443,219
frogs (5)	2474	203	215	182,726,077
lamport (8)	867	70	49	62,669,266
mcs (5)	896	77	50	60,556,458
mcs (6)	12	7	7	332,544
phils (6)	422	36	27	14,348,901
phils (7)	2103	196	125	71,933,609
phils (8)	1613	105	70	43,046,407

of main memory is acceptably small. To parallelize bitstate hashing on multiple CPU cores, the set of successors is partitioned, and all partitions are scanned in parallel. In bitstate hashing, a bit set is never cleared. As we conduct BFS, state caches with different replacement strategies are also feasible.

19.5 Experiments

The GPU model checker CuDMoC uses a bitstate table with 81,474,836,321 entries consuming 9.7 GB of RAM. Models are taken from the BEEM library. Table 19.1 analyses the performance of the GPU algorithm compared to the CPU's. We used the `-deviceemu` directive of the nvcc compiler to simulate the experiments on the CPU (we found no significant speed difference between simulating the code with this directive and converting it by hand to, e.g., POSIX threads). The table shows that using the GPU for the successor generation results in a mean speed-up (sum of all 1 Core + CPU times / sum of all 1 core + GPU) of 22,456 / 2,638 = 8.51. Column *8 Core + GPU* displays additional savings obtained by utilizing all eight CPU cores for duplicate detection, operating simultaneously on a partitioned vector of successors. The comparison demonstrates only the influence to the whole model checking process; larger speed-ups were reached by considering only this aspect.

In order to compare CuDMoC with the current state-of-the-art in (multi-core) explicit-state model checking, we additionally performed experiments on the (most recent publicly available) releases of the DiVinE (version 2.2) and Spin (version 5.2.4) model checker. DiVinE instances were run with `divine reachability -w N protocol.dve` with N denoting the number of cores to use and aborted when more than 11GB RAM was used. Table 19.2 shows the comparison in running time of the one core and the eight-core versions. Of course, DiVinE is not able to check some instances due to its exhaustive duplicate detection it needs to store all visited states in full length, which is less memory efficient than bitstate hashing. One interesting fact in the frogs (5) protocol is that DiVinE is only able to verify this instance in single-core mode. We assume that the queues, needed to perform communication between the cores, consume too much memory. Additionally, we display the number of reached states to indicate the number of states omitted. In the largest instance, the amount of states omitted is at most 3%. The speed-up averaged over all successful instances is 3,088 / 863 = 3.58 for the one core and 632 / 484 = 1.31 for the eight-core implementation (DiVinE naturally utilizes all cores for expansion, while CuDMoC uses the additional cores only for duplicate checking).

Spin is also able to manage an exhaustive representation of the closed list; however, due to the memory limitations of an exhaustive search, we decided to compare CuDMoC against Spin

Table 19.2: Comparing the GPU model checker with DiVinE. Times given in seconds, o.o.m denotes out of memory.

	CuDMoC			DiVinE		
Protocol	1 Core	8 Core	States	1 Core	8 Core	States
anderson (6)	25	20	18,206,914	75	21	18,206,917
at (5)	36	29	31,999,395	118	33	31,999,440
at (6)	170	119	160,588,070	674	189	160,589,600
bakery (7)	30	28	29,047,452	95	26	29,047,471
bakery (8)	250	182	253,111,016	–	-	o.o.m
elevator (2)	30	23	11,428,766	74	21	11,428,767
fisher (3)	10	9	2,896,705	12	3	2,896,705
fisher (4)	7	7	1,272,254	5	1	1,272,254
fisher (5)	126	86	101,027,986	541	141	101,028,339
fisher (6)	16	13	8,321,728	37	10	8,321,728
fisher (7)	555	360	386,281,613	-	-	o.o.m
frogs (4)	20	17	17,443,219	69	15	17,443,219
frogs (5)	203	215	182,726,077	787	-	182,772,126
lamport (8)	70	49	62,669,266	238	68	62,669,317
mcs (5)	77	50	60,556,458	241	68	60,556,519
mcs (6)	7	7	332,544	0	0	332,544
phils (6)	36	27	14,348,901	122	36	14,348,906
phils (7)	196	125	71,933,609	768	-	71,934,773
phils (8)	105	70	43,046,407	405	-	43,046,720

Table 19.3: Comparing the GPU model checker with Spin and bitstate storage. Times given in seconds. Column Speed shows the quotient states/time. Protocol mcs 5 was aborted after 10 hours, having generated 6,308,626.

	CuDMoC			Spin Bitstate BFS		
Protocol	1 Core	Speed	States	1 Core	Speed	States
anderson (6)	25	728,276	18,206,914	26	698,282	18,155,353
anderson (8)	669	804,923	538,493,685	228	618,216	140,953,300
at (5)	36	888,872	31,999,395	40	790,811	31,632,471
at (6)	170	944,635	160,588,070	146	727,404	106,201,110
bakery (7)	30	968,248	29,047,452	29	942,202	27,323,870
bakery (8)	250	1,012,444	253,111,016	156	78,283	12,212,250
elevator (2)	30	380,958	11,428,766	19	601,239	11,423,554
fisher (3)	10	289,670	2,896,705	4	724,170	2,896,681
fisher (4)	7	181,750	1,272,254	2	636,131	1,272,262
fisher (5)	126	801,809	101,027,986	141	614,026	86,577,752
fisher (6)	16	520,108	8,321,728	13	639,997	8,319,972
fisher (7)	555	696,002	386,281,613	242	547,841	132,577,710
frogs (4)	20	872,160	17,443,219	19	916,191	17,407,634
frogs (5)	203	900,128	182,726,077	136	853,619	116,092,290
lamport (8)	70	895,275	62,669,266	8	917,817	7,342,543
mcs (5)	77	786,447	60,556,458	–	–	0
mcs (6)	7	47,506	332,544	1	36,598	36,598
phils (6)	36	398,580	14,348,901	43	333,412	14,336,722
phils (7)	196	367,008	71,933,609	229	297,427	68,110,830
phils (8)	105	409,965	43,046,407	139	304,714	42,355,353

with bitstate hashing Spin has two options for performing reachability, BFS and DFS. Table 19.3 presents the results in BFS, which has no multi-core implementation. Spin experiments were performed by calling `spin -a protocol.pm; cc -O3 -DSAFETY -DMEMLIM=12000 -DBFS -DBITSTATE -o pan pan.c; ./pan -m10000000 -c0 -n -w28;`. For the sake of clarity, we also present the number of reached states for both model checkers. We see that the number of states varies extremely for the larger instances. We explain the diversity with the size of the bitstate tables (in Spin 2^{28} = 268,435,456 entries were chosen, as we could not increase this size because the remaining memory was occupied by the algorithm). CuDMoC achieves an average speed of 637,279 compared to Spin with an average speed of 593,598. Although the speed-up is not significant, we highlight the fact that CuDMoC stores all the reached states on external memory for later usage, while these states are lost in Spin. Storing the information on external storage in Spin leads to a slowdown by a factor of 2 and more.

As the Spin BFS algorithm is not parallelizable, we were forced to compare our implementation to the DFS version of Spin and bitstate hashing called via `spin -a protocol.pm; cc -O3 -DSAFETY -DMEMLIM=8000 -DBITSTATE -DNCORE=N -DNSUCC -DVMAX=144 -o pan pan.c; ./pan -m10000000 -c0 -n -w27;` (for one core), and `./pan -m10000000 -c0 -n -w25;` (for eight cores) with N denoting the

Table 19.4: Comparing the GPU model checker with Spin and partial state storage; time in seconds, speed in states/second.

	CuDMoC		Spin Bitstate					
Protocol	8 Core	Speed	1 Core	Speed	States	8 Core	Speed	States
anderson (6)	20	910,345	58	313,911	18,206,893	9	2,017,465	18,157,188
anderson (8)	440	1,223,849	1316	275,800	362,954,000	78	1,859,341	145,028,600
at (5)	29	1,103,427	90	355,547	31,999,291	12	2,630,998	31,571,983
at (6)	119	1,349,479	399	339,403	135,422,110	42	2,476,482	104,012,280
bakery (7)	28	1,037,409	48	573,577	27,531,713	8	3,413,837	27,310,696
bakery (8)	182	1,390,719	456	488,071	222,560,800	39	3,062,315	119,430,320
elevator (2)	23	496,902	47	243,165	11,428,769	8	1,427,956	11,423,654
fisher (3)	9	321,856	7	413,815	2,896,707	2	1,448,344	2,896,689
fisher (4)	7	181,750	2	636,128	1,272,256	1	1,272,298	1,272,298
fisher (5)	86	1,174,744	275	367,375	101,028,340	36	2,397,127	86,296,605
fisher (6)	13	640,132	20	416,086	8,321,730	4	2,079,982	8,319,929
fisher (7)	360	1,073,004	1372	281,557	386,296,530	63	2,098,240	132,189,170
frogs (4)	17	1,026,071	26	670,893	17,443,221	5	3,472,759	17,363,799
frogs (5)	215	849,888	289	632,427	182,771,630	24	3,878,232	93,077,570
lamport (8)	49	1,278,964	17	431,974	7,343,562	3	2,447,541	7,342,625
mcs (5)	50	1,211,129	81	358,055	29,002,474	14	2,343,949	32,815,294
mcs (6)	7	47,506	0	–	36,600	0	–	36,948
phils (6)	27	531,440	26	387,130	10,065,395	17	843,330	14,336,624
phils (7)	125	575,468	351	183,494	64,406,569	51	1,217,002	62,067,145
phils (8)	70	614,948	12	766,795	9,201,551	35	1,043,143	36,510,039

number of cores. Table 19.4 shows the running times and per node efficiencies for the tested protocols. implementations are identical in table 19.3; we present only the values for the eight-core implementation here. As we can see, the eight-core implementation of the DFS algorithm is always faster than the CuDMoC implementation. A closer inspection of the number of the visited states reveals that the number of cores has an impact on the size of the bit-state table, thus resulting in different amounts of visited states. In the Anderson (8) protocol, which is the largest checked protocol, CuDMoC identifies 538,493,685 unique states, while the Spin 8 core implementation reaches 145,028,600 states, omitting nearly 70% of the state space. Additional observations showed that at the beginning of the search the speed is higher, since new states are reached more often, than at the end, where many reached states have already been explored.

19.6 Summary

Explicit-state model checking on the GPU has the potential for growing towards an exciting research field. Therefore, we presented a model checker for efficient state space generation for featuring explicit-state model checking on the GPU. In the algorithm design we successfully attacked two causes of bad CPU performance of the model checker: transition checking and successor generation and exploited a GPU-friendly representation of the model. Bitstate-based duplicate detection has been delayed for and parallelized on the CPU. The results show noticeable gains, likely to rise on more advanced GPU technologies.

Of course, improving the speed-up is still subject to further research. For example, computing the hash values may be executed in parallel on the GPU, while generating the successor states. We restricted our exposition to BFS. As other algorithms discussed in literature like best-first search for directed model checking may also be streamed, they may be executed on the GPU.

So far, the model checker works on a modern but ordinary personal computer. The presented algorithm can, however, be extended to computing clusters, e.g., by storing the search on shared external space, dividing a BFS layer into partitions, and expanding them on different nodes of the cluster. For this case, however, duplicate checking has to be synchronized. To lift the analyses to full LTL, one can attach GPU breadth-first search to *semi-external model checking*. Together with large RAIDs of hard or solid-state disks, this results in a high-performance LTL model checker, exploiting the current cutting edges of hardware technology.

19.7 Bibliographic Notes

The GPU's rapid increase in both programmability and capability has inspired researchers to map computationally challenging, complex problems to it. These efforts in general purpose programming on the GPU (also known as GPGPU or $(GP)^2U$ programming) have positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computing. Since the memory transfer between the graphics card and main board (on the express bus) is extremely fast, GPUs have helped to speed-up large-scale computations like sorting [301, 443], and computationally intensive applications like folding proteins [365], simulating bio-molecular systems [513] or computing prefix sums [321].

GPUs are directly programmable in C using the Compute Unified Device Architecture Cuda. Instead of the delayed elimination of duplicates for supporting large-scale analyses on disk [225], we consider RAM-based model checking with Bloom filters [56] in the form of (double) bitstate hash tables [346]. State space construction via BFS is the essential step and the performance bottleneck for checking large models [593, 84, 35, 434, 245, 633, 202]. Besides checking safety properties, variants of BFS generate LTL property counterexamples of minimal length [279]. Moreover, BFS is the basis for distributed (LTL) model checking algorithms like OWCTY [529] and MAP [82] as well as for constructing perfect hash functions from disk in *semi-external model checking* [218].

Explicit graph algorithms utilizing the GPU (with a state space residing in RAM or on disk) were presented, e.g., by [319]. In model checking, however, state space graphs are generated implicitly, by the application of transitions to states, starting with some initial state. Additionally, considering the fundamental difference in the architectures of the processing units, solutions developed for multi-core model checking [345], however, hardly transfer to the GPU. For state-space generation on the GPU in search challenges like sliding-tile puzzles, large speed-ups with respect to single-core CPU computation [227] were established. Based on computing reversible minimal perfect hash functions on the GPU, one-bit reachability and one-bit BFS algorithms were proposed. Specific perfect hash functions were studied. In solving Nine-Men-Morris a speed-up factor of over 12 was obtained. Specialized hashing for ranking and unranking states on the GPU and a parallel retrograde analysis on the GPU was applied. Unfortunately, the AI exploration approaches hardly carry over to model checking, as general designs of invertible hash functions, as available for games, are yet unknown.

While [225] pioneered explicit-state model checking with delayed duplicate detection on the GPU by accelerating state set sorting, in the meantime there have been several attempts to exploit the computational power located on the graphics card. In all other GPU-based model checking algorithms we are aware of; however, the state space is generated on the CPU. For example, GPGPU-based probabilistic model checking [66] boils down to solving linear equations via computing multiple sparse matrix-vector products on the GPU. The mathematical background is parallelizing Jacobi iterations. While the PCTL probabilistic model checking approach accelerates one iterated numerical operation on the GPU, for explicit-state LTL model checking we perform a single scan over a large search space. As a result, we introduce a conceptually different algorithm, suited to parallel model checking of large models.

Barnat et al. [34] presented a tool that accelerates LTL model checking. They adjusted the MAP algorithm to the GPU to detect the presence of accepting cycles. As in bounded model checking [50], the state space was generated in layers on the CPU, before being transformed into a matrix representation to be processed on the GPU. The speed-ups are visible, but the approach is limited by the memory available on the GPU and able to check properties in moderately sized models only.

Different options have been proposed to increase coverage [346], including the choices of hash functions, e.g., from a set of universal ones (our state hash functions borrowed from Rasmus Pagh [501] are universal). To increase space utility, cache-, hash-, and space-efficient Bloom filters have been proposed [524] and compress a static dictionary to its information-theoretic optimum by using a Golomb code. They have not been extended to the dynamic case of breadth-first model checking. Moreover, hashing implementation refinements like sequential chaining and hash compaction techniques are available.

Chapter 20
Computational Biology

In this chapter we solve the multiple sequence alignment problem, a combinatorial challenge in computational biology, where several DNA, RNA, or protein sequences interpreted as strings are to be arranged for high similarity. The proposal applies randomized Monte-Carlo tree search with nested rollouts and can improve the solution quality over time. Instead of learning the position of the letters, the approach learns a policy for the position of the gaps. The Monte-Carlo beam search algorithm we use has a low memory overhead and can be invoked with constructed or known initial solutions. Experiments show promising results in improving existing alignments.

20.1 Introduction

Multiple[1] sequence alignments (MSAs) are frequently used for the analysis of DNA, RNA, or protein sequences in order to determine the evolutionary relation between species with a common ancestor, to predict the so-called secondary/tertiary structure, as well as the functional centers, in which as few possible mutations as possible occur (assuming that similar sequences inherit similar structures and function).

Algorithmically, MSA boils down to the cost-optimal alignment of strings. Smaller problems can be solved optimally, and the dynamic programming solution relates to approximate string matching. We invoke fixed-memory-bound randomized search that incorporates no expert knowledge in form of refined heuristics, but a series of random walks (rollouts) to learn a mapping (policy) for sampling the search space. The algorithm can improve over existing solutions and incorporates initial alignments into the search. As other algorithms are memory-bound, with its low memory profile it can serve as an add-on over existing approaches.

We start with a concise formulation of the MSA problem. Given a set of n sequences $S = \{s_1, s_2, \ldots, s_n\}$ with $s_i \in \Sigma^*$ for all $i = 1, 2, \ldots, n$, and Σ being a final alphabet. A *sequence alignment* (of length k) consists of a set of n sequences $A = \{a_1, a_2, \ldots, a_n\}$ with $a_i \in \Sigma'^*$ for all $i = 1, 2, \ldots, n$, where $\Sigma' = \Sigma \cup \{\text{“}-\text{”}\}$ and $\text{“}-\text{”} \notin \Sigma$. For each aligned sequence $a_i \in A$ we have length $|a_i| = k$. If all letters “$-$” are removed from $a_i \in A$, we get back s_i. For $n = 2$, the alignment is *pairwise*, for $n > 2$ *multiple*. A *gap* G consists of a single or a sequence of letters $g = \text{“}-\text{”}$. Moreover $gaps\_num(a_i)$ is the *number* of empty letters in the aligned sequence $a_i \in A$ and $|G|$ the *length* of gap G. Particularly we have $|G| = 1$ for $G = \langle g \rangle$ and letter g is located at *position* $gap\_pos_i(g)$ in sequence $a_i \in A$.

For DNAs the alphabet Σ_{DNA} is $\{A, G, C, T\}$ denoting the nucleobases adenine, guanine, cytosine and thymine. For RNA the nucleobase uracil, abbreviated by U, is used instead of thymine, so that $\Sigma_{RNA} = \{A, G, C, U\}$. The protein alphabet contains 20 amino acids.

[1] This chapter is based on joint work with Zhihao Tang, and Peter Kissmann It puts together and improves the work from [229, 205].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_20

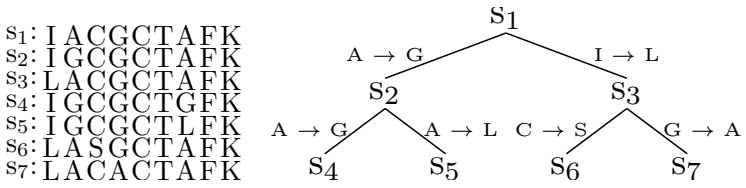


Figure 20.1: An MSA and its phylogenetic tree.

In an alignment all sequences are written on top of each other such that the number of columns with matching letters is maximized. Gaps are inserted to slide letters in the alignment. A substitution occurs if two different letters meet; a gap is a deletion and/or an insertion of a letter and called *indel*. The assumption is that the alignment with the least number of indels is biologically most plausible.

Figure 20.1 shows an example of a protein MSA with $n = 7$ having no gaps, and the according phylogenetic tree where internal nodes denote the ancestor sequences, where I (Isoleucine), L (Leucine), F (Phenylalanine), K (Lysine) and S (Serine) are the one-letter abbreviations for the amino acids. To judge the quality of an MSA an evaluation function is required.

An *evaluation* is a function $F : A \to \mathbb{R}$. For a pairwise alignment $A = \{a_1, a_2\}$ with $a_i = \langle c_{i1} c_{i2} \dots c_{ik} \rangle$ and $c_{ij} \in \Sigma'$, $i = 1,2$ and $j = 1,2,\dots,k$, its evaluation is the sum of similarities f of all alignment columns $F(A) = F(a_1, a_2) = \sum_{j=1}^{k} f(c_{1j}, c_{2j})$. For an MSA $A = \{a_1, a_2, \dots, a_n\}$ the evaluation $F(A)$ is defined as the sum of values for all sequence pairs, i.e.,

$$F(A) = F(a_1, a_2, \dots, a_n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} F(a_i, a_j).$$

Let $\mathscr{A}$ be the set of all MSAs that can be generated by a set of sequences $S = \{s_1, s_2, \dots, s_n\}$. The *optimal MSA* $A^\star \in \mathscr{A}$ is an MSA with $F(A^\star) = \min_{A \in \mathscr{A}} F(A)$, if the evaluation is based on distances or $F(A^\star) = \max_{A \in \mathscr{A}} F(A)$. Given a set of sequences $S = \{s_1, s_2, \dots, s_n\}$, the *MSA problem* is to find the optimal MSA for $A^\star$ for S.

For a set of sequences more than one optimal MSA may exist (Figure20.2) yielding different biological explanations. All solutions have the same edit-distance 4. $F(A)$ can calculate not only the similarities (maximum problems) but also the dissimilarities (minimum problems).

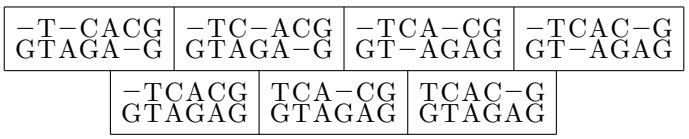


Figure 20.2: Two sequences with 7 optimal MSAs.

We consider *affine gap costs* where *gap opening* has cost op and *gap extension* cost ex (per extension), so that gap G has total cost $P(G) = op + ex \cdot |G|$.

For n sequences of maximal length q, standard *dynamic programming* (DP) computes an optimal solution with memory $O(q^n)$ and time $O(2^n \cdot q^n)$, so that alternative algorithms are required.

The algorithm iterative-deepening dynamic programming (IDDP) combines dynamic programming with iterative-deepening A* on the graph representation of the DP matrix. It expands edges not nodes. A lower bound $h(e)$ is devised based on precomputed pattern database of triples. We have $f(e) = g(e) + h(e)$, so that $f(e)$ for an edge e is the estimated cost of a path of the start edge to reach the end edge via the current edge

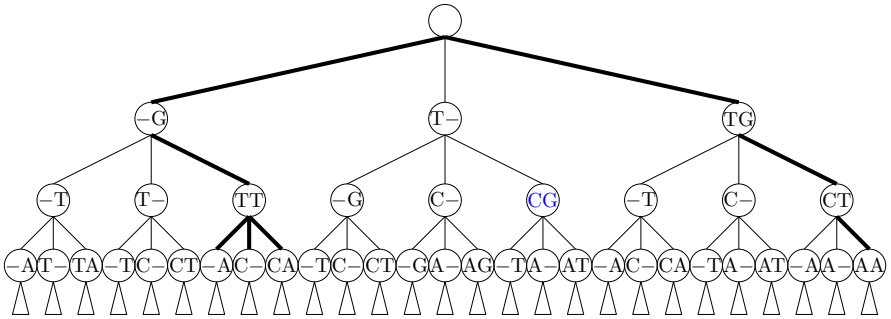


Figure 20.3: The search tree for a sample pairwise alignment.

e IDDP inherits the advantages of DP and IDA*; it has a fixed ordering so that every node is visited once and includes a lower bound for guidance.

20.2 Monte-Carlo Tree Search for MSA

Monte-Carlo search denotes a class of randomized tree search algorithms that has been designed for search spaces with large node branching factors and weak evaluation functions. By learning the proper choice of successors over time they can converge to the overall optimal solutions.

The intuitive method for the MSA problem is to enumerate possible alignments and after evaluating them, to choose the best one. The search tree can be constructed by a sequence of decisions and solved via NRPA and BeamNRPA. We study two possible encodings.

We assume that each letter v in Σ' has a fixed location $index(v)$, so that for a string $V = \{v_1, v_2, \ldots, v_n\}$ in Σ'^* we obtain $index(V) = \sum_{i=1}^{n} index(v_i) \cdot |\Sigma'|^{n-i}$, where n is the length of V and $|\Sigma'|$ the size of the alphabet.

20.2.1 Construction of all Alignment Columns

An MSA consists of columns. Every column is a string in Σ'^n. In the search tree we generate, the root represents an empty node and all other nodes a column in the alignment. Thus, an MSA corresponds to a path from the root to the leaf (see Figure 20.3; optimal MSAs of Figure 20.2 have bold edges).

During the construction the first step is to recursively enumerate all possible strings that may appear in this column (see Algorithm 20.2). The depth of the tree is n as all strings have to have the same length. In each level for every letter of an alternative string s_i we have a) if all letters have been inserted then the following columns are labeled by a gap; b) if there are remaining letters that have a fit, then they are inserted into the MSA and the position i in this column either is the corresponding letter in s_i or a singleton-letter gap. Additionally, the number of all alternative strings is returned. Temporary variables $char\_idx[i]$ store how many letters have already been inserted to s_i.

In this model we learn which string should appear in which column. The maximal length of an MSA is the sum of all input strings. A policy in this model is a mapping $(\sum_{i=1}^{n} |s_i|) \times |\Sigma'|^n$ where $|s_i|$ is the length of s_i.

A random MSA is constructed in Algorithm 20.1. Exploiting the policy, a string is randomly chosen. The variable *align_idx* represents which column is currently constructed. With the variable and the index of an alternative string, we can access the policy value and determine the probability of choosing it. The last step is to

Algorithm 20.1: Constructing an alignment according to policy.

```
procedure alignment_col(alignment, policy)
char_idx ← {1,...,1}
align_idx ← 1
col ← alignment.start
repeat
  col.num ← enumeration(col.alternatives, char_idx, 0)
  sum = 0.0
  for i ← 1 to col.num do
    value[i] ← exp(policy[align_idx][col.alternatives[i]])
    sum ← sum + value[i]
  r ← rand([0,...,sum])
  i ← 1
  sum ← value[1]
  while sum < r do
    i ← i + 1
    sum ← sum + value[i]
  col.index ← col.alternatives[i]
  transform the index col.alternatives[i] to the corresponding
  string
    and save in col.string
  for i ← 1 to n do
    if col.string[i] is not a gap then
      char_idx[i] ← char_idx[i] + 1
  align_idx ← align_idx + 1
  col ← col.next
until all sequences are read through
return alignment
```

Algorithm 20.2: Enumerate all possible alignments.

```
procedure enumerate(A, char_idx, seq_idx)
if seq_idx = 1 then
  static num ← 0
  static str ← {0,0,...,0}
if seq_idx ≤ n then
  if char_idx[seq_idx] > |s_seq_idx| then
    str[seq_idx] ← the index of the gap character
    enumerate(A, char_idx, seq_idx + 1)
  else
    str[seq_idx] ← the index of the gap character
    enumerate(A, char_idx, seq_idx + 1)
    str[seq_idx] ← the index of the char_idx[seq_idx]-th
      character in sequence s_seq_idx
    enumerate(A, char_idx, seq_idx + 1)
else
  num ← num + 1
  transform the string str to the corresponding index
    and save in a[num]
return num
```

update the variables to prepare for the next column. The steps are repeated until all letters have been inserted, so that all columns are constructed and stored in a list. At the end, the MSA is evaluated and returned.

The enumeration process is recursive, starting with $seq\_idx = 0$ and ending with $seq\_idx = n$. As the transformation reads a string of length n, the worst case of Algorithm 20.2 takes $T_{enum}(n) = 2 \cdot T_{enum}(n-1)$ steps

Algorithm 20.3: Multiple sequence alignment with gaps.

```
procedure alignment_gap(alignment, policy)
for seq_idx ← 1 to n do
  alignment.gaps_num[seq_idx] ←
  alignment.length − |s_seq_idx|
  alignment.is_gap[seq_idx] ← {FALSE, ..., FALSE}
  for gap_idx ← 1 to alignment.gaps_num[seq_idx] do
    sum ← 0.0
    for pos ← 1 to alignment.length do
      if ¬alignment.is_gap[seq_idx][pos] then
        value[pos] ← exp(policy[seq_idx][gap_idx][pos])
        sum ← sum + value[pos]
      else
        value[pos] ← 0.0
    r ← rand([0, ..., sum])
    pos ← 1
    sum ← value[1]
    while sum < r do
      pos ← pos + 1
      sum ← sum + value[pos]
    alignment.gaps_pos[seq_idx][gap_idx] ← pos
    alignment.is_gap[seq_idx][pos] ← TRUE
  /* sort alignment.gaps_pos[seq_idx] or not */
return alignment
```

with $T_{enum}(0) = O(n)$. This induces $T_{enum} = O(n \cdot 2^n)$. We see that the time for constructing a column is equal to $T_{col} = T_{enum} + 2 \cdot O(2^n) + 2 \cdot O(n) + O(1) = O(n \cdot 2^n)$. Moreover, as we use the sum-of-pairs evaluation we get $T_{eval} = C_n^2 \cdot k = O(k \cdot n^2)$, where k is the length of the sequence alignment. Together we have $T_{colalign} = k \cdot T_{col} + T_{eval} = O(k \cdot n \cdot 2^n + k \cdot n^2) = O(q \cdot n^2 \cdot 2^n)$, with $k = n \cdot q$ being the worst case, and q being the maximal length of all sequences.

20.2.2 Construction of all Alignment Gaps

A sequence alignment is fully determined by the position of gaps. Based on this state representation idea for each sequence s_i the policy is stored as a matrix of size $gap(a_i) \cdot k$, where $gap(a_i)$ is the number of gap letters in the aligned sequence a_i and k the length of the alignment. Again, Monte-Carlo tree search is used to learn, where a gap letter is present in which column of the alignment.

If the length of the alignment is known, the number of gap letters can be determined upfront. Then the positions of all gaps letters can be chosen one after the other. The temporary variable *is_gap* helps to determine all legal gap positions. The algorithm is executed for all sequences until the entire MSA can be evaluated. After all gaps in one sequence are done, we can sort them (line 20) which has pros and cons.

We avoid gap-only columns by moving the gap in the longest sequence to $gap\_pos_{new} = (gap\_pos_{org} + (-1)^i \cdot \lfloor (i+1)/2 \rfloor) \ mod \ k$, $i = 1, 2, 3, \ldots$ (see Figure 20.4). We check that there are no gap-only columns left. If no satisfying position can be found, the original one is maintained. Algorithm 20.3 does, however, not cover this special case. Alternatively, we may allow gap columns, as they do not change the score.

The running time of this model is easy to analyze. A random alignment is constructed one by one. Sequence s_i contains $k - |s_i|$ gap letters. We obtain $T_{gapalign} = O(\sum_{i=1}^{n} \sum_{j=1}^{k-|s_i|} (2k)) + T_{eval} = O(q^2 \cdot n^3)$, with $k = n \cdot q$ in the worst case and q being the maximal sequence length.

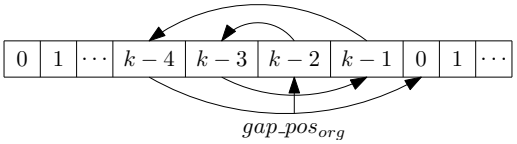


Figure 20.4: Resolving gap-only columns.

A−CGG
A−−TG
ATCGG

ACGG
A−TG

A−−TG
ATCGG

Figure 20.5: Sample MSA projections.

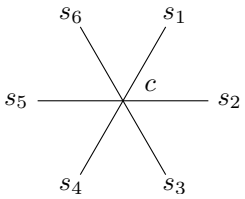


Figure 20.6: A star alignment tree of sequences $\{c, s_1, \ldots, s_6\}$.

20.2.3 Construction of an Initial Alignment

In the second model, prior knowledge is requested in the form of the length of the optimized alignment. This information can be supplied by the user or via an initial alignment.

Let $S = \{s_1, \ldots, s_n\}$ be a set of sequences and S' a subset of S. Assume $A_S = \{a_1, \ldots, a_n\}$ to be an MSA of S. The *projection* of A_S wrt S' is the MSA $proj(A_S, S')$, constructed as follows

- all rows in A_S that do not correspond to sequences in S' are removed
- all columns that only contain gap letters are removed.

If $A_{S'} = proj(A_S, S')$, where $A_{S'}$ is an MSA of S', we say that A_S is compatible with $A_{S'}$.

An example for $S = \{$"ACGG", "ATG", "ATCGG"$\}$, $S' = \{$"ACGG", "ATG"$\}$ and $S'' = \{$"ATG", "ACTCGG"$\}$ is shown in Figure 20.5. We see an MSA A_S of S, a projection $proj(A_S, S')$, and another projection $proj(A_S, S'')$.

An *alignment tree* for a set of sequences S is a labeled tree. In this tree the node set is S and every edge (i, j) is labeled by the optimal pairwise alignment of two sequences s_i and s_j.

In an alignment tree the relations between all sequence pairs are represented. There are different options for constructing such a tree. We consider the special case of the tree being star-shaped (Figure 20.6).

The algorithm for constructing an initial MSA has two stages. The basis is a set of precomputed pairwise alignments (see Algorithm 20.4). For each pair of sequences (s_i, s_j) the distance to the optimal alignment is computed. For each sequence s_i all distances of the optimal alignment corresponding to s_i are added. The sequence with the minimal total distance is chosen as the center, all other sequences are leaves.

The second stage is to construct an MSA based on the pairwise alignment stored at the edges. Whenever an MSA of the sequences $\{c, s_1, \ldots, s_i\}$ is constructed, the optimal pairwise alignment of c and s_{i+1} is inserted. This insertion preserves the rule *once a gap — always a gap*. Therefore, the constructed MSA is compatible with all pairwise alignments in the alignment tree. For example, $c =$ "ATGCATT", $s_1 =$ "AGTCAAT" and $s_2 =$ "ACTGTAATT". The alignments of c and s_1 or c and s_2 are

Algorithm 20.4: Constructing an initial alignment.

```
procedure initial_alignment()
for i ← 1 to n do
    for j ← i+1 to n do
        compute the optimal alignment of s_i and s_j with distance
        d_*(s_i, s_j).
for i ← 1 to n do
    total[i] ← 0
    for j ← 1 to n do
        total[i] ← total[i] + d_*(s_i, s_j)
c ← arg min_i total[i]
choose an arbitrary sequence s ∈ S \ {s_c}
let A be the optimal pairwise alignment of s_c and s
S' ← {s_c, s}
while S' ≠ S do
    choose an arbitrary sequence s ∈ S \ S'
    combine A with the optimal pairwise alignment of s_c and s
    S' ← S' ∪ {s}
return A
```

$$a = \mathrm{ATG{-}CATT} \qquad a' = \mathrm{A{-}TGC{-}ATT}$$
$$a_1 = \mathrm{A{-}GTCAAT} \quad \text{and} \quad a'_2 = \mathrm{ACTGTAATT}$$

In the second alignment we find a gap prior to letter 'T' in d sequence a'. According to the golden rule the gap in a'' is preserved. Through the combination from a and a' we can generate $a'' =$ "A–TG–C–ATT", so that the final MSA is

$$a'' = \mathrm{A{-}TG{-}C{-}ATT}$$
$$a''_1 = \mathrm{A{-}{-}GTC{-}AAT}$$
$$a''_2 = \mathrm{ACTG{-}TAATT}$$

The MSA is not optimal as we could substitute a''_2 by "ACTGT–AATT". It is, however, a good approximation.

A similarity cost function f is *proper* if 1) for all $x \in \Sigma'$, we have $f(x,x) = 0$; 2) for all $x,y,z \in \Sigma'$, we have $f(x,z) \leq f(x,y) + f(y,z)$. Assuming a *proper* similarity cost function f, and d being the column sum of f, a set of sequences $S = \{c, s_1, \ldots, s_n\}$ and a star alignment tree T with center c. If $A = \{a, a_1 \ldots, a_n\}$ is an MSA of S with length k that is compatible with all optimal alignments in T, then for all $1 \leq i, j \leq n$ we have $F(a_i, a_j) \leq F(a_i, a) + F(a, a_j) = F_*(s_i, c) + F_*(c, s_j)$.

We consider column r in MSA A. According to the second property of a proper cost function for an arbitrary letter $b \in \Sigma'$ we have $f(a_i[r], a_j[r]) \leq f(a_i[r], b) + f(b, a_j[r])$. If $b = a[r]$, we have $f(a_i[r], a_j[r]) \leq f(a_i[r], a[r]) + f(a[r], a_j[r])$. The distance of a pairwise alignment is the sum of distances of all columns. Thus,

$$\begin{aligned} F(a_i, a_j) &= \sum_{r=1}^{k} f(a_i[r], a_j[r]) \leq \sum_{r=1}^{k} \big(f(a_i[r], a[r]) + f(a[r], a_j[r]) \big) \\ &= \sum_{r=1}^{k} f(a_i[r], a[r]) + \sum_{r=1}^{k} f(a[r], a_j[r]) = F(a_i, a) + F(a, a_j). \end{aligned}$$

Following the assumption we have that the MSA A is compatible with all optimal alignments in T. Therefore, the projections of A wrt $\{s_i, c\}$ are optimal alignments of s_i and c. Following the first property of a proper cost function, we have $f(-,-) = 0$, so that the distance of a pairwise sequence alignment does not change if an only-gap column is removed. Hence, $F(a_i, a) = F_*(s_i, c)$, and $F(a, a_j) = F_*(c, s_j)$. □

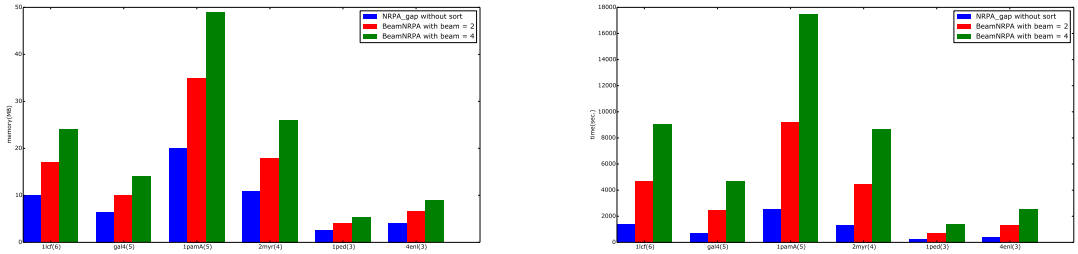


Figure 20.7: Space (top) and time needed by (Beam)NRPA.

Let $S = \{s_1, \ldots, s_n\}$ be a set of sequences, f be a proper similarity cost function, F be the column sum of f, and $A = \{a_1, \ldots, a_n\}$ be an MSA of S, constructed via Algorithm 20.4. Then, $F(a_1, \ldots, a_n) \leq \left(2 - \frac{2}{n}\right) \cdot F_*(s_1, \ldots, s_n)$.

We assume that MSA $A^\star = \{a_1^\star, \ldots, a_n^\star\}$ is optimal for S, i.e., $F(a_1^\star, \ldots, a_n^\star) = F_*(s_1, \ldots, s_n)$, and $c = s_n$ is the center. We compute the distance between A and $A^\star$.

$$\begin{aligned} F(a_1, \ldots, a_n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} F(a_i, a_j) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} F(a_i, a_j) \leq \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \left(F_*(s_i, c) + F_*(c, s_j)\right) \\ &= \frac{1}{2} \left(\sum_{i=1}^{n-1} \sum_{j=1}^{n-1} F_*(s_i, c) + \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} F_*(s_j, c)\right) \\ &= \frac{1}{2} \left(\sum_{j=1}^{n-1} \sum_{i=1}^{n-1} F_*(s_i, c) + \sum_{j=1}^{n-1} \sum_{i=1}^{n-1} F_*(s_i, c)\right) = (n-1) \cdot \sum_{i=1}^{n-1} F_*(s_i, c) \end{aligned}$$

and

$$\begin{aligned} F(a_1^\star, \ldots, a_n^\star) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} F(a_i^\star, a_j^\star) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} F(a_i^\star, a_j^\star) \\ &\geq \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} F_*(s_i, s_j) = \frac{1}{2} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} F_*(s_i, s_j)\right) \\ &\geq \frac{1}{2} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} F_*(c, s_j)\right) = \frac{1}{2} n \cdot \sum_{j=1}^{n} F_*(c, s_j) = \frac{1}{2} n \cdot \sum_{i=1}^{n} F_*(s_i, c) \end{aligned}$$

Therefore, we have

$$\frac{F(a_1, \ldots, a_n)}{F_*(s_1, \ldots, s_n)} = \frac{F(a_1, \ldots, a_n)}{F(a_1^\star, \ldots, a_n^\star)} = \frac{(n-1) \cdot \sum_{i=1}^{n-1} F_*(s_i, c)}{\frac{1}{2} n \cdot \sum_{i=1}^{n} F_*(s_i, c)} = 2 - \frac{2}{n}. \square$$

The MSA that is constructed via the star-shaped alignment tree is, therefore, an upper bound for the distance of the optimal MSA.

20.3 Experimental Results

BAliBASE is a library of biological alignments that optimize an informal biological *meaning*. Having a formal sum of pairwise scores on BAliBASE entries cannot replace a comparison with bioinformatics competitors.

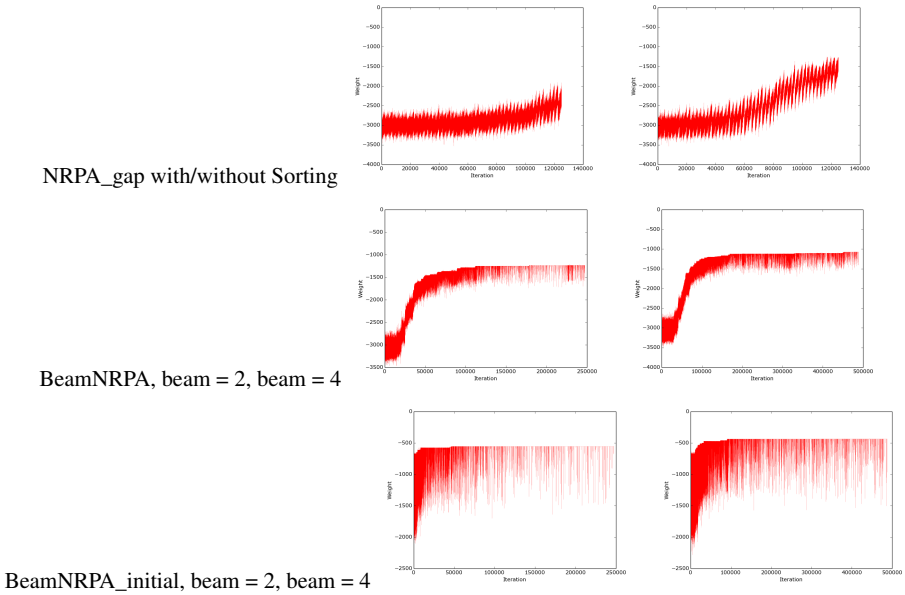


Figure 20.8: Learning curves of *1ped*.

However, our interest is showing the potential of Monte-Carlo search for the MSA problem in terms of saving space and post-hoc optimization.

Reference 1 consists of 82 sequence groups, partitioned into nine classes according to the length (short, medium, long) and similarity (large, medium, small). Among those we chose *test 3*, consisting of 28 sequence groups with three to six sequences of different similarity. From the set of MSAs we chose 1ped and 4enl (three sequences) and 1lcf (six sequences), together with the groups 2myr (four), ga14 (five), and 1pamA (four), which are supposedly the hardest. The implementation supports FASTA and MSF formats. The web presentation comes with manual close-to-optimal solutions.

For these sequence groups at most 20 MB RAM was allocated, which is by far lower than the one in IDDP and variants. On the other hand, BeamNRPA was better than NRPA: the wider the beam, the better the solution. The number of rollouts for BeamNRPA is $beam \cdot iteration^{level}$ (we allow a beam width other than 1 only in level 1), and chose $beam = 1,2,4$, $iteration = 50$ and $level = 3$. BeamNRPA with $beam = 1$ is NRPA. The initial alignment is defined by the star algorithm and improved by the optimizer.

In NRPA_col a policy is a matrix of size $(\sum_{i=1}^{n} |s_i|) \times |\Sigma'|^n$, so that the memory requirements are exponential in n. This leads NRPA_col to fail for five or six sequences and to bad results in many others. For NRPA_gap a policy is a matrix of size $(k - |s_i|) \times k$ for every s_i, so that memory requirements are polynomial in $|s_i|$ and k. Only four of 28 groups needed more than 10 MB space, and 20 MB was the overall maximum. For DP and its variants the space complexity is $O(|s_1| \cdot \ldots \cdot |s_n|)$. A biological sequence (DNA/protein) may have over one thousand bases/amino acids. Hence, the memory requirements are huge. The algorithm saves only the positions of all gaps in an alignment. Obviously, the number of gaps is much less than the length of an aligned sequence. Therefore, the required memory in the program is small.

Sample learning curves for 1ped and 1pamA are shown in Figures 20.8 and 20.9. NRPA_gap without sorting often resulted in a better quality than with sorting, where *1pamA*, *2myr* and *1lcf* are the only exceptions. Thus, we used no sorting in BeamNRPA. Memory and time performances of NRPA and BeamNRPA are cross compared in Figure 20.7. The wider the beam, the higher the computational cost. On the other hand, as shown in Figure 20.8, the larger the search tree, the better the solution found by BeamNRPA.

If the initial alignment could be improved after determining the alignment, we called the adaptation function 10 times (α-value of 1) to come up with an initial policy. For the sequence group *1ped* an alignment better than

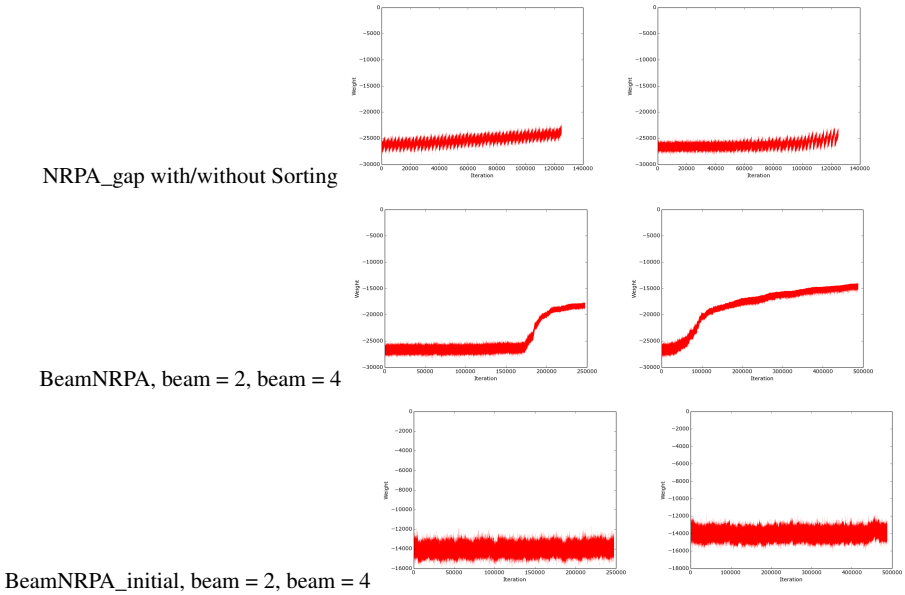


Figure 20.9: Learning curves of *1pamA*.

the initial one was found quickly (see Figure 20.8). The initial alignment of *1pamA* had a score of minus 8,291. Unfortunately, for this hardest group BeamNRPA did not improve much within the given parameter range (see Figure 20.9).

Finally, we optimize solutions from the BAliBASE benchmark with BeamNRPA_gap. The results show an improvement (with respect to the cost function) in 20 groups, equal results in six groups (1ac5, 1bgl, 1dlc, 1fieA, 1gpb, 1gtr) and worse result in two groups (1pamA and 1taq). Altogether there are 28 sequence groups. For the groups "1pamA" and "1taq" the program cannot return a better solution than BAliBASE (from beam = 2 and 4). For these six groups (1ac5, 1bgl, 1dlc, 1fieA, 1gpd and 1gtr) the program returns the same good solutions as BAliBASE (from beam = 2 and 4). For the other 20 groups the better solutions are found from beam = 2 or 4 (beam = 2 sometimes can return a better solution than beam = 4).

20.4 Summary

In this chapter we applied Monte-Carlo tree search for the multiple sequence problem. The results for learning gaps with BeamNPRA are promising. The approach has a low memory overhead, can be used from scratch and for post-hoc optimization. With respect to the cost function, we found improvements to BAliBASE alignments.

It is possible to improve the policy representation by learning inter-dependencies of gap positions within the set of sequences. A further yet unexplored option is the parallelization of BeamNRPA. The advantage of BeamNRPA is that it is easier to parallelize as all policies in the beam can be read and updated concurrently. It has the additional feature that it can be parallized in every level of the search. As the number of iterations is usually larger than the number of threads, the searches in each thread are iterative. Another option to deal with concurrency issues in the parallelization is to use low-level compare-and-swap.

20.5 Bibliographic Notes

Computational biologists have declared the MSA problem to be a *holy grail* [315]. One reason is that solving this problem often leads to a high memory demand, which has been partially leveraged with frontier search [338, 420], refined heuristics, and variants of memory-limited [675, 674] or iterative-deepening heuristic search [562]. Most of these approaches provide strategies to limit exploring the search space induced by dynamic programming [42]. Tools like Clustal(W/Omega) and Blast compute approximate MSAs with probabilistic models. IDDP has been proposed by [562]. A partial expansion alternative to IDDP has been proposed and parallelized by [326].

Known computational biology algorithms for the MSA problem are Clustal-Omega [575], MUSCLE [235, 236], and MAFFT [387]. For a rising number of sequences, the MSA problem is NP-hard [640]. Precursory work showed considerable scaling but often neglects biological relevant features like the inclusion of similarity cost matrices and affine gap costs. Exceptions are iterative deepening dynamic programming [562], its externalization [205], and a search variant using partial expansion [326]. Still, the memory requirements rise exponentially with the problem complexity (measured in the sum of the input sequences). The algorithm that we chose has successfully been used for vehicle routing [106, 194].

In [105] it has been said that parallelizing NRPA is involved, since the policy has to be shared among the threads. However, a series of optimization problems have been solved, e.g., TSPs with Time Windows [538, 108, 106] and Morpion Solitaire [105, 545].

The evaluation function by [446] is used to compute *edit distances*. For DNA alignment we support scoring matrices used in WU-BLASTN [335] and FASTA [506], and for protein alignment the PAM (Point Accepted Mutation) matrix [149, 60], the PET91 matrix [370], and BLOSUM (BLOck SUbstitution Matrix) [334]. An algorithm to construct an initial alignment automatically is described in [431].

BeamNRPA_gap with initial alignment

	beam = 2				*beam* = 4			
	len	score	time	mem	len	score	time	mem
1ajsA	457	-2663	2126	11M	459	-2680	4169	15M
1cpt	468	-937	1669	9792K	467	-828	3313	12M
1lvl	501	-2027	1915	11M	502	-1961	4117	13M
1pamA	730	-11736	12350	53M	728	-11896	23831	73M
1ped	402	-556	722	5128K	402	-430	1447	6664K
2myr	598	-4788	6150	26M	595	-4501	11504	37M
4enl	425	-892	997	6124K	427	-903	1959	8228K
gal4	492	-4643	4813	22M	488	-4342	8832	30M
1ac5	551	641	3084	13M	545	802	6090	19M
1adj	432	3210	479	9552K	429	3392	964	9552K
1bgl	1072	1958	7248	47M	1071	3890	13190	47M
1dlc	655	2555	2550	18M	654	2615	5029	18M
1eft	419	1355	957	8572K	417	1440	1888	8576K
1fieA	702	5565	1147	22M	703	5567	2250	22M
1gowA	542	1138	1975	12M	542	1225	3925	15M
1pkm	474	1809	834	10M	473	2081	1652	10M
1sesA	494	2917	2238	16M	488	3379	4390	16M
2ack	561	-509	3557	20M	556	-215	7039	22M
arp	490	435	3209	15M	488	622	6341	20M
glg	553	2568	3222	19M	551	2620	6376	19M
1ad3	464	5133	611	10M	463	5121	1210	10M
1gpb	877	17561	4097	53M	878	17578	7891	53M
1gtr	466	7671	1162	16M	465	7658	2289	16M
1lcf	799	2330	8778	58M	799	3392	17135	58M
1rthA	565	8897	1120	23M	563	9022	2202	23M
1taq	978	1889	7947	62M	977	1879	15483	62M
3pmg	619	6744	2006	16M	620	6731	3936	16M
actin	416	7883	824	13M	416	7916	1622	13M

BeamNRPA_gap for BAliBASE optima (BBO)

	BBO	*beam* = 2				*beam* = 4			
	score	len	score	time	mem	len	score	time	mem
1ajsA	-1292	449	-1264	1698	9852K	449	**-1258**	3378	13M
1cpt	520	461	558	1397	8440K	461	**602**	2750	10M
1lvl	-750	516	-750	2284	12M	516	**-720**	4522	16M
1pamA	**-2366**	677	-5252	8715	39M	678	-3290	17215	54M
1ped	-42	398	-15	647	4548K	396	**40**	1274	5956K
2myr	-1490	554	-1561	4018	21M	554	**-1452**	8048	28M
4enl	-336	441	-293	1164	7428K	438	**-265**	2298	9804K
gal4	-876	439	-811	2168	11M	438	**-779**	4283	15M
1ac5	**2375**	524	**2375**	2141	11M	524	**2375**	4247	15M
1adj	4037	421	4064	200	2192K	421	**4087**	395	2556K
1bgl	**7394**	1002	**7394**	2263	11M	1002	**7394**	4505	15M
1dlc	**4906**	638	**4906**	1733	9724K	638	**4906**	3419	11M
1eft	2211	412	**2257**	921	6100K	412	**2257**	1831	7232K
1fieA	**6815**	689	**6815**	640	4300K	689	**6815**	1279	5628K
1gowA	2710	546	2712	2112	11M	545	**2730**	4135	15M
1pkm	2981	468	2981	617	4252K	468	**2984**	1231	5428K
1sesA	5896	465	5896	1086	6488K	465	**5907**	2167	8520K
2ack	3470	536	**3473**	2321	11M	536	**3473**	4542	15M
arp	3875	450	3889	1492	8556K	450	**3891**	2974	11M
glg	4959	514	5007	1502	9268K	513	**5109**	2937	10M
1ad3	5409	459	5415	491	4100K	459	**5426**	982	4752K
1gpb	**20141**	854	**20141**	2605	12M	854	**20141**	5145	17M
1gtr	**8807**	451	**8807**	665	4320K	451	**8807**	1321	5660K
1lcf	25001	747	25007	4168	19M	747	**25015**	8268	26M
1rthA	10400	556	**10475**	788	4940K	556	10472	1575	6336K
1taq	**13545**	949	13048	5222	25M	949	13300	10482	34M
3pmg	7867	589	**7869**	956	6080K	589	7868	1899	7912K
actin	8489	415	**8556**	793	5108K	415	8530	1575	6620K

Chapter 21
Logistics

The dynamics and complexity of planning and scheduling processes in groupage traffic require efficient, proactive, and reactive system behavior to improve the service quality while ensuring time and cost-efficient transportation. At first, we, therefore, implement a multiagent system to emerge an adequate system behavior and focus on the decision-making processes of agents that is based on the Traveling Salesman Problem (TSP) with aspects like contract time windows, individual restricted capacities of trucks, premium services and varying priorities of dynamically incoming orders. We present an optimal depth-first branch-and-bound asymmetric TSP solver with constraints on tour feasibility and depot reachability at any step of the process. We use established benchmarks as well as its inclusion in a real-life multiagent-based simulation. Simulated scenarios are based on real customer orders and are applied on real-world infrastructures. The results reveal that efficient optimal decision making in multiagent systems increases the service quality and meets the requirements and challenges in logistics.

In this chapter we also look at packing problems that naturally arise in container loading. Given a set of 3D ISO-oriented objects and a container, the task is to find a packing sequence of the input objects consisting of the ID, location, and orientation that minimizes the space wasted by the packing. Instead of the decision problem, we look at the packing optimization problem, minimizing the total height of a packing. The solution uses extreme points and applies Monte-Carlo tree search with policy adaptation. The implementation is considerably simple and conceptually different from mathematical programming branch-and-bound and local search approaches. Nonetheless, the results in solving 2D and 3D packing problems are promising.

21.1 Introduction

Vehicle Routing One[1] challenge is to optimize the planning and controlling processes in groupage traffic to improve the service quality while ensuring cost- and time-efficient transport processes. To meet the high requirements on complexity and dynamics, we implement a multiagent system to ensure a flexible system behavior and develop efficient and optimal decision-making algorithms for participating agents.

In transport logistics the decision making often relies on efficient solutions to the *Traveling Salesman Problem* (TSP), a touchstone for many general approaches in combinatorial optimization. In our application of a forwarding agency, the TSPs are generated via shortest path reductions of route networks. Each order to be served corresponds to one city in the TSP.

Due to one-way streets, we consider *asymmetric TSPs* (ATSPs). Many ATSP solvers consider the according *Assignment Problem* (AP), which can be solved by the Hungarian algorithm or refined approaches, followed

[1] This chapter is based on joint work with Max Gath, Moritz Rohde, Ashraf Abdo, Christoph Greulich, Malte Humann, Otthein Herzog, and Michael Lawo. It puts together and improves the work from [197, 196, 1].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_21

by some tour patching strategies. ATSPs can be converted into *symmetric TSPs*, but this requires doubling the number of cities. Empirical TSP exploration results often partially refer to the 8th DIMACS Challenge.

Many additional constraints apply in practice. Besides capacity and time constraints we have a certain mix of *pick-up and backhaul* and *premium service constraints* to be served in the tour, while other non-premium services are optional (but should be maximized). We also consider TSPs with *delivery and backhauls*; not to be mixed with *delivery and pick-up*. With the mix of premium and non-premium tasks we generate a preference problem that includes a combination of hard and soft constraints.

In the *vehicle routing problems* with capacity, time and premium services constraints, backhauling and dynamic change, determining the optimal solution quickly becomes intractable.

Packing Industrial robots have eventually found their way into container packing and unpacking. Such intelligent packaging robots pack and unpack containers with loose packages fully automatically. Some companies have such systems already operating in their logistics centers.

A robot packing system (shown in Figure 21.1) consists of a chassis beneath the robot, a telescope conveyor, a 3D laser scanner, and an interchangeable gripper system. The robot is positioned on the chassis, which is connected to the conveyor belt. This can be extended mechanically and transports robot and chassis into the container. That way the work envelope of the robot will be extended, and the gripper arm of the robot can reach any point in the container. By use of the 3D laser scanner possible gripping positions for the pile of object can be analyzed and an optimal unloading sequence and collision-free trajectories can be computed. The removed packages are then transported by the conveyor belt until the container is completely empty.

In this chapter we look at the inverse problem of optimally packing 3D objects into a container to maximize its load. For loading, the same hardware as outlined above can be used. Compared to the unloading, the loading process is simpler in handling, since the sizes of the goods and the respective positions for pick-up and drop-down are defined. But still there are big challenges especially for the loading of parcel with unequal sizes and weights in containers or swap-bodies.

Whereas parcels are mostly imported in containers they are often distributed in swap-bodies. Here the maximum height of the pile is relevant. DHL, for instance, ordered a maximum stacking height of two meters to reduce accidents when dropping goods. Optimizing the loading process to achieve a compact and solid pile is a requirement for the effective and safe transport and subsequent unloading process.

Our industrial example is finding a tight packaging of objects into one or several containers (see Figure 21.2). Due to the ever-increasing globalization of trade, the amount of cargo shipped in containers is on the rise and the efficient utilization of transportation capacity will be a competitive advantage for companies. Using less containers for shipping a specific number of goods will reduce the costs, not only because of the lower number of containers required, but also because potential breakage is reduced through the dense packing. If the available storage is limited, deciding which items to defer to a later shipment is also an important issue regarding profit and customer satisfaction, e.g., by not violating deadlines. But the problem of packing small items into large objects is not necessarily limited to loading containers and trucks. The large objects can range from pallets to ships and planes or even warehouses. Depending on the large receiving object, the set of small items may also include said pallets and containers, irregularly shaped items like furniture, or just small rectangular boxes. The general problem can be considered a cutting and packaging problem and, therefore, also covers material utilization when cutting glass or wood for example. It is even possible to use this concept for scheduling problems by redefining the axis appropriately. Assuming a limited number of workers and machines that require uninterrupted processing time for a specific order, cutting and packing methods can be used to schedule workers and machines to reduce the downtime when neither enough workers nor machines are available.

The 3D packing problem is a true extension of the rectangle packing problem, already known to be a hard optimization problem. Even the bin packing problem is known to be NP-complete. While pseudo-polynomial time algorithms have been derived for the 1D packing problem, no such results have been derived for the 3D packing problem. We study the packing problem that maximizes the volume utility of a single container with and without orientation restrictions on the boxes. A solution is an ordering of (possibly oriented) objects together with a set of coordinates. We enforce objects to form a connected arrangement of objects. Objects are only

stable if their center of mass rests on top of another object. We assume that unoccupied space can be filled with other material so stability is not a concern for this study.

Figure 21.1: Container packing scenario with mobile robot platform and conveyor belt.

Formally, we are given a set of rectangular boxes $B = \{b_1, \ldots, b_n\}$ and a rectangular container C. Let x_i, y_i and z_i represent the three dimensions of the box b_i, and X, Y, and Z represent the three dimensions of the container. The objective of the problem is to select a subset $S \subseteq B$ and assign a position to each box $b_j \in S$ in the 3D space of container C such that $\sum_{b_j \in S} x_j \cdot y_j \cdot z_j$ is maximized subject to the constraints that all boxes must be totally contained in C, no two boxes intersect in 3D space and every edge of the packed box must be parallel or orthogonal to the container walls.

If oriented boxes are to considered, the values of triples (x_j, y_j, z_j) can be permuted. Given that for all j the values of x_j, y_j, and z_j are pairwise different, for the 2D problem we have $2! = 2$ possible orientations, while in 3D we have $3! = 6$ different orientations of an object (suppose an unoriented package has dimension $1 \times 2 \times 3$; consequently, there are six different orientations that lead to the following rotated dimensions for this package: $1 \times 2 \times 3$, $2 \times 1 \times 3$, $1 \times 3 \times 2$, $2 \times 3 \times 1$, $3 \times 1 \times 2$, and $3 \times 2 \times 1$). In the 3D *strip packing problem* that we consider in this chapter, we are given a set of 3D rectangular items and a 3D open box B. The goal is to pack all the items in B such that the height of the packing is minimized. We consider the most basic version of the problem, where the items must be packed with their edges parallel to the edges of B. In the oriented case, we allow rotation. A trivial upper bound $X_{\max}$ for the height of the packing is $\sum_{i=1}^{n} x_i$. Of course, more effective bounds can be derived.

The problem of the center of mass of a box is apparent. It is dependent on the package weight distribution and whether additional support *bridges* can be inserted to support overhanging parts. The problem of packing many small boxes into a single larger box is part of a number of cutting, packing, scheduling, and transportation applications. There are several heuristic solvers, but the progress in exact solvers that can deal also with orientation, in general, and integer programming solvers, in particular, has been limited.

21.2 Dispatching in Groupage Traffic

In *groupage traffic*, several orders to different destinations with less-than-truckload (LTL) shipments are served by the same truck to decrease total cost. In pickup tours, trucks transport loads from their origin to a local depot where the shipments are consolidated to build economical loads. Through LTL networks the load is transported to a depot in the destination area where each good is delivered to its destination. The process planning complexity is even increased by individual qualities of shipments like weight, volume, priority, and value. Handling

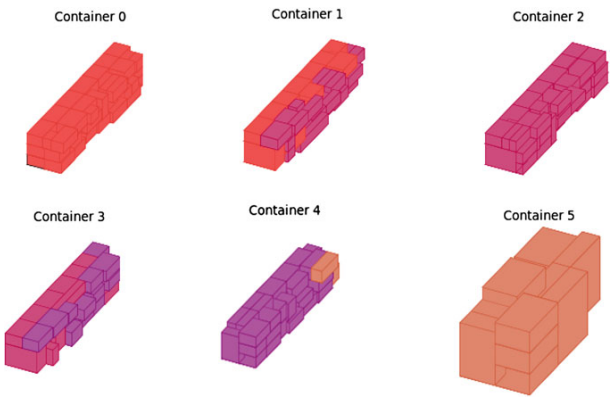


Figure 21.2: Container packing solution.

the complexity is aggregated by the high degree of dynamics that result also from unexpected events, such as an exact amount and properties of shipments are not known in advance.

Quality of service is an important factor to succeed in the economic objectives. The transportation of so-called premium services must be guaranteed with respect to their time windows while considering other hard constraints, e.g., the capacity of vehicles. In order to increase service quality through short transit times and reliable deliveries it is mandatory to handle the high degree of dynamics and complexity of logistic processes with adaptive, reactive system behavior.

Regarding the dispatching processes each vehicle has to find a tour with minimum costs, such that each pickup and delivering stop is visited exactly once and the tour returns to a central depot. This problem can be described by a TSP with stops i, $i \in \{1,2,\ldots,n\}$. All distances between two stops i and j are given by $c_{i,j} \in \mathbb{R}^+$ with $c_{i,i} = 0$ for $1 \leq i,j \leq n$. Feasible solutions are permutations of $(1,2,\ldots,n)$ with the additional constraint that the first and the last city to visit is the depot. Real transport infrastructures are commonly represented by directed graphs, so that we search an optimal tour for an ATSP. In logistic transport networks participating forwarding agencies must pay high amounts of penalty if they are not fulfilling the agreed commitments. Therefore, we distinguish hard premium service constraints that must be delivered on date of receipt and soft non-premium service constraints that can be delayed by up to two days.

Pickup and delivery of *premium services* is mandatory and defined by

$$p_i = \begin{cases} 1, \text{ if } i \text{ is a premium stop} \\ 0, \text{ otherwise} \end{cases} \tag{21.1}$$

Hence, the priority of premium stops is higher than that of other stops.

The optimal tour of the asymmetric TSP must be feasible and fulfill the following requirements ordered by their priorities for the variables:

$$x_{i,j} = \begin{cases} 1, \text{ if } (i,j) \text{ is part of the tour} \\ 0, \text{ else} \end{cases} \tag{21.2}$$

1. Maximize the number of transported premium services: $max \sum_{i=1}^{n} \sum_{j=1}^{n} p_i \cdot x_{i,j}$
2. Maximize the number of visited stops: $max \sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j}$
3. Minimize the total cost of the path: $min \sum_{j=1}^{n} \sum_{i=1}^{n} c_{i,j} \cdot x_{i,j}$ subject to
   a. $\sum_{i=1}^{n} x_{i,j} = 1$ for all $j \in \{1,\ldots,n\}$;
   b. $\sum_{j=1}^{n} x_{i,j} = 1$ for all $i \in \{1,\ldots,n\}$;
   c. $x_{i,j} = \{0,1\}$ for all $j,i \in \{1,\ldots,n\}$;

d. $\sum_{j\in S}\sum_{i\in S} x_{i,j} \leq |S|-1$ for all $S \subseteq \{1,\ldots,n\}$.

We assume that all premium stops must be traversed such that we have to find the tour with minimum costs that includes all premium services and the maximum number of stops while satisfying all time and capacity constraints. Therefore, the problem changed into a maximizing-minimizing problem.

21.2.1 Constraint ATSP Solving

To apply *Branch-and-bound* (BnB), we extend depth-first search (DFS) with upper and lower bounds. To determine a lower bound for the ATSP, we transform it into the *Assignment Problem* (AP), which can be solved with the *Hungarian algorithm* in $O(n^3)$. While the AP is a relaxation of the asymmetric TSP, it can be used as a heuristic function for the ATSP. In order to solve the AP, we extend the cost matrix with the distance from the depot to the current node (we want to return to the central depot which is not necessary the starting point). After solving the AP, we subtract it to compute the current lower bound. For increasing the upper bound (which is the sum of the weights in the subgraphs) we determine the minimum value to merge the subgraphs of the AP by comparing the respective columns and rows in the cost matrix and choosing the arcs with minimum costs.

An initial upper bound can be obtained by constructing any solution, e.g., established by a greedy approach. Unfortunately, for larger TSPs the branching process consumes a lot of time to determine a greedy solution. Therefore, we additionally computed the upper bound U at each node by applying Karp-Steel patching. As with standard DFS, the first solution obtained might not be optimal. With *depth-first BnB* (DFBnB), however, the solution quality improves over time together with the global value U until eventually the lower bound $L(u)$ at some node u is equal to U. In this case an optimal solution has been found, and the search terminates.

A *Constraint ATSP* is an ATSP in which additional state constraints are applied. States are only discarded from the search if they do not fulfill the constraints. This weakens but does not invalidate the lower bound. For example time and capacity constraints as well as priorities must be satisfied.

For time constraints, the due date d_i for latest pickup (or delivery) at each stop $i \in \{1,\ldots,n\}$ has to be met. Each stop may require additional individual processing time δ_i (e.g., for loading), which can be compiled away (the solution value then changes by $\sum_{i=1}^{n} \alpha_i$). In some cases, release dates r_i, $i \in \{1,\ldots,n\}$ are given, at which the order at stop i becomes issued. If the arrival is too early, we must wait for time β_i.

For the TSPTW we choose the net time for traversing the edges to be minimized rather than the total time, which, nonetheless, is progressed for adaption to release and check with due dates. The travel time between stops π_i and π_j on tour π is denoted by $t_{i,j}$, $i,j \in \{1,\ldots,n\}$. Additional *capacity constraints* for the vehicle yield a *Capacitated TSP*.

Let w_i be the freight weight at stop i and Δ_w be the maximum weight of the vehicle. A tour π with stops $\{\pi_1,\ldots,\pi_n\}$ and depot d is *feasible* for the capacitated TSPTW if for all $i = 1,2,\ldots,n$ we have $\sum_{l=1}^{i}(\alpha_i+\beta_i+t_{i-1,i}) \leq d_i$, $\sum_{l=1}^{i} w_i \leq \Delta_w$, and d $= \pi_n$.

The pseudo-code is shown in Algorithm 21.1. The procedure is invoked with the number of cities n the depot d, cost 0 and upper bound U set to some reasonable estimate (U can obtained using some heuristics; the lower it is, the better the pruning, but in case no upper bound is known, it is safe to set U to ∞). The tour is maintained globally and updated during backtracking. Another global variable *best* keeps track of the actual solution path. DFBnB sorting the set of successors according to increasing L-values is an optional refinement to the algorithm that often aids in accelerating the search for finding an early solution.

Release and due dates in TSPTWs induce a precedence relationship at each city. The relationship implies a partial ordering, so that a city can be selected as a successor of city only if it does not violate the imposed ordering. Given a bitvector of a cities visited so far the subsumption check for precedence can be executed by native Boolean operations in $O(1)$. Similarly, premium services are checked on the word level.

Algorithm 21.1: DFBnB for Constraint TSPs.

```
DFBnB(n,u,g,U)
  tour_depth(u) ← u
  if (depth(u) = n − 1)
    if (g + c_u,d < U)
      best ← tour; U ← g + c_u,d
  else
    for each v_j in nextcities(u)
      if Constraint(v_j)
        if (g + h(v_j) < U)
          call DFSBnB(n, v_j, g + c_u,v_j, U)
```

If no pruning was taking place, every possible solution would be generated, so that the optimal solution would eventually be found. Sorting of children according to the L-values has no influence on the algorithm's completeness. The condition $L(v_j) < U$ confirms that the node's lower bound is smaller than the global upper bound. Otherwise, the search tree is pruned, as for admissible weight functions exploring the subtree cannot lead to better solutions than the one stored with U. All further pruning rules (like precedence or premium service pruning) cut off infeasible branches from the search tree so that the solution will be optimal for the TSP problem looked at. Precedence pruning retains optimality for the TSPTW, while capacity pruning remains optimal for capacitated TSP, etc. This shows that Algorithm 21.1 is optimal for an admissible lower bound, (wrt the above pruning rules).

With h_a we define the heuristic that is derived from solving AP. The Hungarian algorithm for computing the solution, as well as lower bound offsets based on the tour being Hamiltonian takes time $O(n^3)$. It can incrementally be computed in $O(n^2)$, but bookkeeping becomes involved. With h_c we define the heuristic that is defined as the sum of the column minima in the distance matrix. If cities are visited, column minima are subtracted from the sum. The travel distance back to the depot is added on top. We compute h_c incrementally in $O(1)$ time and space. Let u be the successor of v, and m_i be the precomputed minima of columns i, $i = \{1,\ldots,n\}$ in distance matrix D. Moreover, let the heuristic value of the depot d be defined as $h_c(d) = \sum_{i=1}^{n} m_{d,i}$. For each expanded node u we compute $h' = h(u) - m_d$ if $u = d$, and $h' = h(u) - c_{u,d}$, otherwise. For each generated successor v of u we have $h_c(v) = h' - m_v + c_{v,d}$.

An advantage of the tree structure is that the constraint checks are available in constant time and space. The current time, capacity and premium service information are stored at each node. Considering the time efficiency of the algorithm, in our implementation we thus rely on the fact that the bitvectors can be realized by one computer word: we have bitvectors for the cities seen, the relative ordering among them imposed by the time window, and the priority service constraints. All bitvector operations (setting, clearing of bits, check for subsumption) run in $O(1)$.

To determine the optimal solution for a TSP with premium services we extend the algorithm. We assume that all premium stops must be traversed so that we have to find the tour with minimal costs that includes all premium services and the maximal number of stops while satisfying all time and capacity constraints. Therefore, the problem changed into a *max-min problem* and branching is not applicable by comparing the costs to the lower bound L. We have to find a feasible solution that includes all premium services within maximum search tree depth and the shortest distance within that depth.

To speed up search for a first solution in depth $n-1$, the computation of the lower bound is disabled and a subtree is cut if the current tour is not a feasible tour. We prune the tree with the following rule using bitvectors and Boolean operations. Before starting the search, we sort the premium services by their capacities. The number of premium services that are included in the best solution is saved and updated if a better solution is found. Next, we compute the difference Δ between the number of currently included premium services and the number of premium services in the best-known solution. Afterwards, the weight of Δ lightest not considered premium services is accumulated, and we check if they exceed the maximum capacity of the truck. The solution remains optimal and complete because a feasible solution with more premium service exists. The pruning rule is applied to regular orders accordingly if all premium services are included within the tour. The application of this pruning rule is optional, because it is done in $O(\Delta)$ time (for summing up the weight of orders not considered).

Nevertheless, it speeds up the algorithm if good solutions are known in advance, the maximum capacity of the truck is reached and no more orders can be operated. In this case big subtrees can be pruned early. If we find a first feasible complete tour objectives 1. and 2. in Def. 21.2 are fulfilled.

Let w be the word width of the computer. The incremental solver for the Asymmetric TSP with Time Windows, Capacity, and Premium Service Constraints and heuristic h_c runs in $O(n/w)$ per node. It allocates $O((n^3/w)$ space in its initialization phase. The space consumed for the distance matrices as well as their compression and copies is $O(n^2)$. The matrices for successor reordering and filtering take $O(n^2)$ and each state on the stack requires $O(n/w)$ space. As the stack is limited by the depth times the number of successors, its memory needs are also bounded by $O((n/w) \cdot n^2)$. All other structures (tours, interval sizes, rows and columns and minima, partial ordering bitvectors, successor sets and visited flags) take at most $O(n)$ space.

After satisfying the premium services, the remaining objective is to reduce the total cost and the problem has changed to a classical TSP with constraints. As a result, the computation of lower and upper bounds described above is activated. In this case, less nodes are expanded because the searching process is goal-directed. If heuristic h_a instead of h_c is activated, the efforts at each node are higher.

21.2.2 Agent Dispatching and Simulation

In our setting agents represent trucks and orders. Whenever a new transport request has to be processed, a new agent is created that represents an order. The goal of the agent is to find a proper transport service provider to pick up or deliver the shipments with respect to the time windows and premium service constraints. The agent starts the contract-net protocol for negotiating with available transport service providers. All operating trucks are represented by an agent as well. The truck agents evaluate the proposals by determining its additional costs for transporting. In order to schedule new orders also while transporting other shipments, the truck has to consider its current capacity constraints and position. For example, picked up shipments reduce the capacities, and the position of the vehicle affects the determination of shortest ways and tours. Consequently, we link the planning and decision-making processes of the agents directly with their execution behaviors and consider all relevant observed changes of the environment as well as the internal state of the agent within the decision making and tour planning. On the other hand, new plans can effect the executed actions of the agents. Therefore, the truck agents check during driving, if the next stop has changed and if necessary he adapts the tour. In real processes as well as in the simulation the handing processes (boarding and deboarding of shipments) must not be interrupted. This requirement is satisfied by not adopting plans that manipulate the running handling processes.

To transport a premium service instead of conventional orders or another premium service by driving a shorter distance, already accepted orders may not be included in the new plan. If these orders have not been boarded the truck agent sends a message to the agent that acts on behalf of the corresponding order. Afterwards, the order agent negotiates with other transport service providers again. Potentially, this results in a series of computation and communication intensive negotiations between agents to achieve small improvements. To weaken this effect (especially if several shipments are processed consecutively within a short time window and the global allocation changes significantly) the agent waits for a certain period before it starts the negotiation procedure. For optimal decision making, agents must solve a TSP for each proposal. Consequently, numerous TSPs have to be solved in the planning and controlling processes.

Applying simulation for evaluating multiagent systems before their deployment in real applications is an accurate cost and time reducing method. The physical world within the simulation environment is modeled as directed graph. Nodes represent traffic junctions or logistic sources and sinks, while edges represent different types of ways, e.g., roads, motorways, trails, and waterways. To model sound planning and controlling processes in the logistic domain, we extended our system to import transport infrastructures from OpenStreetMap (OSM) databases. The directed graph includes information about the real-world speed limits, the distance as well as the type of an edge. Particularly within large infrastructures, determining the shortest path between nodes is an essential, costly, and time-consuming procedure within the decision-making process of an agent. However,

```
search(level, iterations, pol)
  best.score = MAXVALUE
  if (level == 0)
    eval = rollout()
    best.score = eval
    best.packing = packing
  else
    policy[level] = pol
    for(i=0; i<I; i++)
      r = search(level - 1,I,pol)
      score = r.score
      if (score < best.score)
        best.score = score
        best.packing = r.packing
        adapt(best.packing,score,level)
    pol = policy[level]
  return best
```

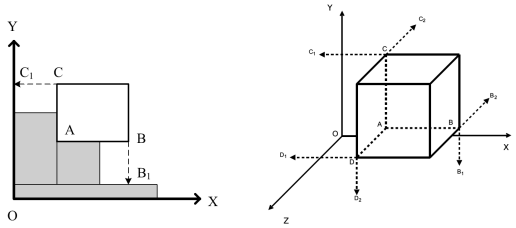


Figure 21.3: NRPA (left), extreme points in 2D and 3D (right)

computing the distance matrix between cities is essential for solving the TSP on a shortest path reduced graph. Computing a distance matrix requires many shortest-path queries with a fixed starting node and is time-critical. Hence, we adapted the search procedure and saved the last visited nodes within a hash map as well as in the heap as long as the start node had not changed. While processing new search requests we check in constant time if the shortest path to the node was already found and retrieve the corresponding node from the heap. Otherwise, shortest path search is continuing at the last expanded node. We extended the search with a cache.

21.3 Container Packing with Nested Monte-Carlo Search

The Nested Monte-Carlo search algorithm is parametrized with the *level* of the search which denotes how deep the search is, and with the number of *iterations*, that shows how strong the policy learning effect within the search is. At each leaf of the recursive search (Level 0), a *rollout* is invoked. A rollout corresponds to a (possibly constraint violating) packing. A rollout performs and evaluates a random run to construct a packing. An example setting is a level-5 search with 50 iterations, which leads to 50^5 rollouts.

Policy adaptation, that changes the probabilities of choosing the successor states in the rollout based on the success of previous experiences. Rather than navigating the tree directly the approach instead uses gradient ascent on the rollout policy at each level of the nested search.

The pseudo-code implementation of NRPA for the packing problem is shown in Figure 21.3 (left). We see that the evaluation of a Level k search relies on the result obtained in Level $k-1$ search and that the results are propagated bottom-up.

21.3.1 Extreme Points

The basic idea of *extreme points* is that when an item j with sizes X_j, Y_j and Z_j is added to a given packing and is placed with the left-back-down corner in position (x_j, y_j, z_j) it generates a series of new extreme points, where additional objects can be placed. The new extreme points are generated by projecting the points with coordinates $(x_j + X_j, y_j, z_j)$, $(x_j, y_j + Y_j, z_j)$ $(x_j, y_j, z_j + Z_j)$ on the orthogonal axes of the container. In 2D this

leads to at most two extreme points generated for each object being placed, while in 3D we have at most six extreme points (see Figure 21.3, right). Each point is projected on all items lying between item i and the wall of the container in the respective direction. If there is more than on item on which a point can be projected, the algorithm chooses the nearest one. Moreover, to avoid wasted space by additionally sliding the object we urge them to respect connectivity and gravity constraints.

If the container is empty, the first object is placed in position $(0,0,0)$ generating extreme points at (X_j, 0,0), (0, Y_j,0), (0, 0,Z_j).

While the extreme point approach applies to floating-point size data, in our experiments we decided to discretize the domains of the object and container sizes to integers. This change supports operations in the integer range progressing a global layout of the objects. The test of intersection and the projection simplifies.

21.3.2 Box Packing

The most important function to be applied in NMC is the *rollout*. Code profiling indicates that most time is spent in this function. In a rollout we randomly walk down the search tree from root to a leaf node to form a complete packing.

The pseudo-code implementation of the rollout function for the packing is depicted in Algorithm 21.2 (left). Using *visited* flags, successors are eliminated from the set of all possible values, so that any generated solution necessarily has to be a permutation of boxes. Some parameters such as *visited*-flags, the set of successors found, the current assignment of packages to container cells, as well as the incrementally generated packing (including ID, location, and orientation) and the (layered) policy tables are kept globally in class member variables. The function *legal* places all remaining objects on all possible extreme points. The outcome is a list of successors m of (possibly oriented) objects together with their coordinates l and orientation o.

To avoid the generation of clearly dominated solutions it also slides objects (in turn) towards lower x-, y- and z-coordinates. This ensures that the packing is connected and (to some extent) stable. As we do not expect knowledge on the weight distribution within an object, we do not compute the center of mass. This constraint can be added by the user or bypassed by adding additional *bridges*.

The number of violations to the enforced constraints can be included into the cost function evaluation that is returned to the NMC algorithm. The major objective of the cost function is to reduce the number of layers in x-direction. As a minor objective, the number of remaining extreme points are minimized. Our implementation features orientation of rectangles.

Objects are *placed* into a one-sided open container of cells. This is done by setting the respective cells to the id of the object. The resulting set of extreme points is computed in the *update* procedure. While objects must fit into the x- and y-dimension of the container, they are allowed to exceed the z-dimension. At the end of the rollout procedure, all cells of the container are *cleared*.

Furthermore, *packing* is a global variable or the parameter and includes the object, its location, and its orientation. Copying of *pol* and *policy* is already done in the search procedure. We need a temporary, which makes the code harder to read.

For the packing problem, adjacencies are less important compared to absolute coordinates. In an existing policy P, rollouts children s' for a node s are chosen wrt $e^{P(s,s')}$. The successors are choosin using a roulette wheel fitness selection based on these values. Initially, all policy values are set to 0. As the entire state-to-state table surely is too big, it is projected to an essential part to be learnt.

Given a packing that improves the current best cost value, policy adaptation (see Algorithm 21.2, right) performs gradient descent as follows. The sequence of children $s' = (s'_0, \ldots, s'_l)$ of states $s = (s_0, \ldots, s_l)$ with $s_{i+1} = s'_i$ has the probability $Prob(s,s') = \prod_{j=0}^{l} e^{P(s_j,s'_j)} / \sum_{i=0}^{l} e^{P(s_j,s'_i)}$. The gradient of the logarithm at j of this term is

Algorithm 21.2: Rollout and policy adaptation functions for the container packing optimization problem.

```
rollout()
  cost = objectsSize = 0;
  nextreme = 1; extreme[0] = 0;
  for (j=0;j<N;j++) visited[j] = false;
  while (objectsSize < N) {
    successors = legal(nextreme);
    sum = 0;
    for (i=0; i<successors; i++)
      value[i] = exp(pol[code(l[i],o[i
          ],m[i])]);
      sum += value[i];
    mrand=random(sum);
    i=0; sum = value[0];
    while (sum<mrand) sum += value[++i];
    object = m[i];
    location[objectsSize] = l[i];
    orientation[objectsSize] = twist[i
        ];
    objects[objectsSize++] = object;
    place(object,twist[i],l[i]);
    visited[object] = true;
    nextreme =
      update(object,twist[i],l[i],
          nextreme);
    if (cost < x+sizes[object].x)
      cost = x+sizes[object].x;
  clear(cost);
  return 1000 * cost + nextreme;
```

```
adapt(packing, cost, level) {
  for (j=0;j<N;j++)
    visited[j] = false;
  object = 0;
  nextreme = 1;
  extreme[0] = 0;
  for(p=0; p<N; p++)
    successors = legal(nextreme);
    object = packing.objects[p];
    l = packing.location[p];
    o = packing.orientation[p];
    layer[level][code(l,o,object)]
        +=
      ALPHA;
    z = 0.0;
    for(i=0; i<successors; i++)
      z += exp(pol[code(l,o,m[i])]);
    for (i=0; i<successors; i++)
      layer[level][code(l,o,m[i])]
          -=
      ALPHA*
        exp(policy[code(l,o,m[i])])/z
          ;
    place(object,o l);
    nextreme =
      update(object,l,o,nextreme);
    visited[object] = true;
  clear(cost);
```

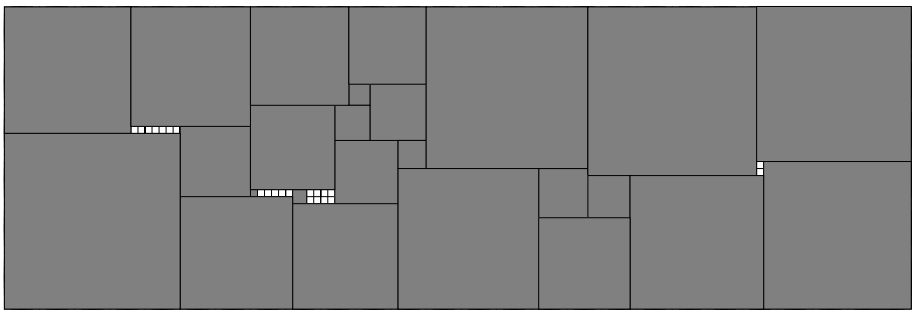


Figure 21.4: Solutions to two square packing problems (for 25 objects). The 1×1 square is removed from the input and assumed to fit.

$1 - e^{P(s_j,s'_j)} / \sum_{i=0}^{l} e^{P(s_j,s_i)}$, so that we add α to the best chosen successor and subtract $\alpha \cdot e^{P(s_j,s'_j)} / \sum_{i=0}^{l} e^{P(s_j,s'_i)}$ from the others, where α is a factor for accelerating the learning process. This ensures that policy adaptation increases the probability of the solution sequence. The policy learned is a mapping from objects together with their orientation to the y- and z-coordinates.

Policies are copied top-down, adapted bottom-up, and improved while progressing from one successor to its sibling.

The solution for packing the squares packing squares 1×1 to 25×25 into a box of size 43×129 problem is shown Figure 21.4. In the implementation, objects are represented by their boundary surfaces. In a discretized occupancy grid representation, this makes intersection tests easier.

Table 21.1: Results in Dumas' TSPTW Benchmark (Exp. nodes and CPU time for the two heuristics are shown).

Prob.	Cost	E_h	T_h	E_c	T_c
n20w20.001	378	49	$< 1s$	2,784,766	$< 1s$
n20w20.002	286	97	$< 1s$	3,234,936	$< 1s$
n20w20.003	394	138	$< 1s$	4,944,477	$< 1s$
n20w20.004	396	156	$< 1s$	2,331,312	$< 1s$
n20w20.005	352	41	$< 1s$	4,017,260	$< 1s$
n20w40.001	254	38,022	3s	103,087,541	18s
n20w40.002	333	88	$< 1s$	11,388,523	2s
n20w40.003	317	1,409	$< 1s$	21,158,796	3s
n20w40.004	388	7,676	1s	35,117,607	6s
n20w40.005	288	10,287	2s	20,801,644	3s
n20w60.001	335	40,810	14s	223,904,879	43s
n20w60.002	244	97,144	7s	81,367,918	15s
n20w60.003	352	399,127	27s	31,292,739	5s
n20w60.004	280	4,055,453	258s	1,245,195,466	238s
n20w60.005	338	105,393	10s	104,049,862	18s
n20w80.001	329	316,992	35s	288,653,549	56s
n20w80.002	338	260,552	36s	166,880,630	33s
n20w80.003	320	15,959	3s	208,526,467	42s
n20w80.004	304	1,258,898	80s	373,077,547	73s
n20w80.005	264	5,224,435	438s	1,660,621,704	332s
n20w100.001	237	1,635,101	52s	1,232,596,799	279s
n20w100.002	222	68,954	7s	2,203,174,867	531s
n20w100.003	310	13,382,035	765s	2,586,795,810	538s
n20w100.004	349	34,289	2s	1,213,551,958	266s
n20w100.005	258	688,887	44s	2,308,713,055	548s

21.4 Experiments

21.4.1 Groupage Traffic

In the first setting we generated a fully-connected random graph of k nodes with edge weights in $[0..C]$. With $C = 10$ our solver can handle problems with 500 cites. For $C = 100$, and up to 100 cities, the experimental outcome shows that there are rare unfortunate cases that need hours for computation, but generally remain tractable. For smaller values of n (the number of cities) and more complex TSPs, our incremental solver with constant time per node was often more effective than computing at each node.

Next, we extended the solver with time windows and focused on the performance of applied heuristics. As release and due times are more difficult to generate randomly, we took an existing benchmark set of TSPTW problems. The results in Table 21.1 examine the search efforts for DFBnB with h_h and h_c. We see that while the number of nodes is substantially lower for h_h, the solving time is sometimes (but not always) larger than for h_c, indicating that the more constrained the problem is, the worse the AP approximation and the better the search with a simpler heuristic. Note that our best results were obtained with a combination of both heuristics, using the more expressive one in the top part of the tree and the less expressive one in the bottom part of the tree.

To investigate the interactions between agents we implemented several scenarios. The transport infrastructure contains 124,462 nodes and 292,521 edges. We started a reverse geocoding process to map the address information to coordinates and determined the nearest neighbor node in the map, to link the addresses with graph nodes. The real weight, premium service constraints, latest delivery times as well as the incoming dates are attached with the order. Since exact delivery times are not available, only the date is considered during evaluation. Thus, we modeled the dynamics by generating new orders successively until all orders of this day have been dispatched. In real transport processes, vehicles with interchangeable units are sent to stops where numerous shipments have to be handled. Consequently, we did not consider orders if more than six orders had to be picked up at the same stop. While the dynamics of the planning and controlling processes of delivery tours are limited

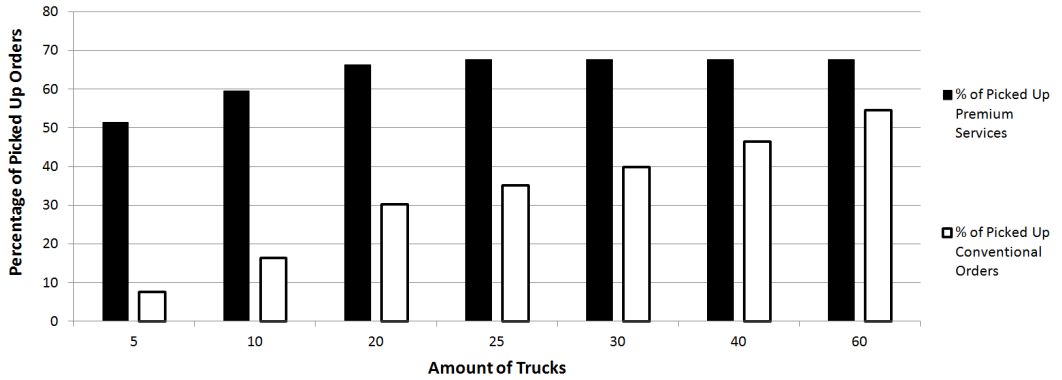


Figure 21.5: Black/White bars show the percentage of picked up premium/conventional services in the selected scenario.

since that shipments have to be loaded before starting the transport processes, we only consider pickup orders and simulate the planning processes simultaneously to the execution of plans.

We modeled seven scenarios. In each scenario 1,265 orders are distributed within a whole week while the number of trucks is varying. About six percent of the total number of modeled orders are premium services. Table 21.2 shows the computation times elapsed during each simulation run for computing the distance matrix with the shortest-path algorithm as well as the time for solving the TSP specified in Def. 21.2. The results reveal that more computation time is required for the reduction of the real-world infrastructure graph to a minimum distance matrix between relevant nodes than for optimally solving of the TSP. Consequently, the shortest path matrix computation is the most expensive part of the decision-making process of the agent, if we are considering the computational effort. Moreover, Table 21.2 indicates that the number of TSPs rises with the number of available trucks. That is obvious, because the TSP solver is an essential part within the decision-making process of each truck.

Table 21.2: CPU time in ms for solving the TSP as well as for shortest path searches during the matrix computations.

#Trucks	#TSP	time for solving TSP in ms	time for matrix computation in ms
5	6,122	199,043	4,912,094
10	11,893	324,566	16,588,955
20	22,457	604,675	56,845,930
25	26,751	972,579	75,882,546
30	31,542	2,246,511	94,921,678
40	40,759	7,979,063	144,742,544
60	56,152	79,704,147	225,749,232

Figure 21.5 shows the percentage of picked-up premium and conventional orders within each scenario. We see that the agent system considers premium services with higher priorities. Even with five trucks nearly 50% of all premium services are sill picked up, while more than 90% of conventional orders are postponed to other days. An increase of the number of available trucks leads to processing more premium services as well as more conventional orders. If about 70% of all premium services were satisfied, only transported conventional orders increased.

21.4.2 Packing

As an indicator for the simplicity of the implementation our single-core C/C++ packing implementation consists of less than 400 LOC. It is generic and supports 2D and 3D non-oriented and oriented packings. The support of higher dimensions would only require different projection and intersection functions. The Java implementation was competitive with the C/C++ program.

21.4.2.1 2D Packing

For 2D rectangle packing, we look at Korf's square packing instances. These combinatorial problems are defined as constraint satisfaction and not as a strip packing minimization problems. Nonetheless, with our general approach we could solve several benchmark problem instances with NRPA using six levels and 20 iterations. The results presented in the form Problem ID : $L \times B$ (time,rollouts) were as follows.

2 : 2×3 (0.1s, 1),
3 : 3×5 (0.1s, 1),
4 : 5×7 (0.1s, 2),
5 : 5×12 (0.1s, 13),
6 : 9×11 (0.1s, 50),
7 : 11×14 (0.1s, 48),
8 : 14×15 (0.2s, 3,728),
9 : 15×20 (0.2s, 8,502),
10 : 15×27 (0.1s, 97),
11 : 19×27 (55s, 3,520,193),
12 : 23×29 (8s, 329,903),
13 : 22×38 (4s, 8,557),
14 : 23×45 (11s, 335,798),
15 : 23×55 (3s, 1,947),
16 : 28×54 (1m31s, 1,658,002),
18 : 31×69 (31.4s, 463,297),
20 : 34×85 (23m15s, 3,483,648), and
22 : 39×98 (49m26s, 4,418,323).

We always took the larger container size value as the undefined one. In contrast to Korf's approach that is based on heuristically guided depth-first branch-and-bound search with a lower bound for computing wasted space based on a reduction to bin packing we leave the container open, and, therefore, do not rely on calculating wasted space.

21.4.2.2 3D Packing

For evaluating 3D packing algorithms, we first generated 3D objects as follows. The sizes of the objects to be packed into a container of size $[1..X_{\max}] \times [1..Y_{\max}] \times [1..Z_{\max}]$ were random choices in $[1..\lfloor X_{\max}/2 \rfloor] \times [1..\lfloor Y_{\max}/2 \rfloor] \times [1..\lfloor Z_{\max}/2 \rfloor]$. To visualize the outcome of the NRPA optimization process in 3D, one packing example is illustrated in Figure 21.6 (left).

In small up to moderately-sized benchmarks, we tested the effect of varying the parameters of the search, namely changing the values level and the iterations in NRPA. Table 21.3 shows the obtained packing quality in terms of the chosen cost function. Shallow search led to smaller runtimes and still good results.

In Figure 21.6 (right) we show four learning curves of an NRPA search with 2 levels, 100 iterations, and up to 200 objects. The experiment with 100 iterations took about half an hour, the one with 150 one hour, and the one with 200 two-and-a-half hours.

n	10	15	20	25	30	35	40	45	50
5,7	4,418.4	6,629.0	8,438.6	9,853.0	11,862.8	14,267.8	15,882.2	18,283.4	20,287.4
4,10	4,420.6	6,633.8	8,639.6	9,847.8	12,659.8	14,283.8	16,082.8	18,682.4	20,694.8
3,22	3,819.2	6,027.6	8,039.6	10,044.4	11,658.2	13,672.6	15,273.2	17,886.4	20,096.0
2,100	3,819.0	5,828.8	7,831.4	9,241.2	12,459.6	12,461.2	16,271.6	16,472.2	19,290.8

Table 21.3: Solution quality in terms of the cost function value obtained by NRPA search for the oriented packing problem with $Y_{\max} = Z_{\max} = 10$ averaged over 5 trials.

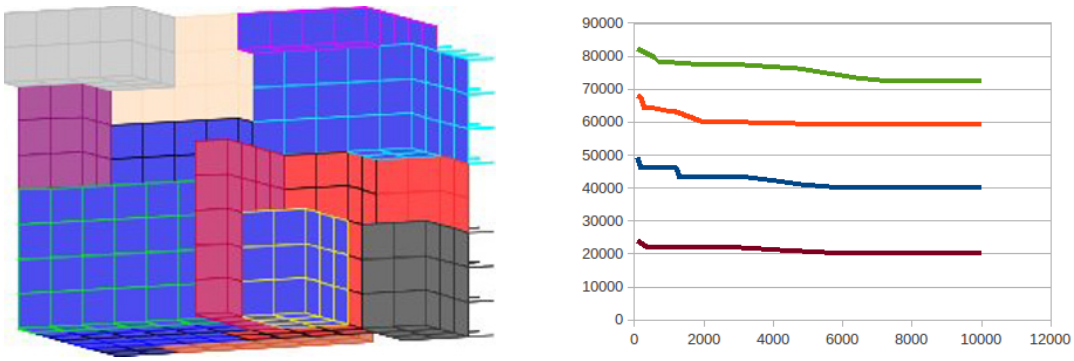


Figure 21.6: Sample 3D packing (left); learning curves for NRPA for a varying number of oriented objects (right, $n = 50, 100, 150, 200$ read from bottom to top). The x-axis is the number of rollouts, the y-axis denotes the obtained solution quality in terms of the cost function.

By reducing the number of rollouts, we could scale the algorithm to optimize packings to 2,000 boxes and more, and, thus that we could handle packing problems for industrial-sized containers. The time complexity for the rollout operation is substantial, so we could only select a small number of rollouts (depending on the time provided). However, this was already sufficient for first optimizations as NRPA is what is called an *anytime* algorithm, which provides a first solution quickly and improves over time.

In an industrial case of 3D container packaging of axis-aligned boxes we even managed to solve an industrial benchmark of 224 rectangular ISO-oriented boxes (in two possible orientations) to be packed in five 40" high-cube and one 20" standard containers, while respecting some additional constraints like some freespace at the door region (see Figure 6.1).

21.5 Summary

We looked at two challenging problems in logistics. First, we have studied a dispatching system matching the requirements of forwarding agencies in groupage traffic. To face the dynamics of consecutively incoming orders and high dynamics, we implemented a reactive and proactive multiagent system. The agents link the planning and scheduling processes directly with their actions. Therefore, changes of the environment can be considered during runtime and induce a reactive behavior. We focused on the planning and decision-making processes of the agents and developed an efficient TSP solver that is crucial for negotiation with service customer agents. The optimal branch-and-bound TSP solver was time and space efficient: it incrementally checks resource, time, and premium service constraints in $O(1)$ time and space per generated search tree node. Moreover, after the allocation of less than a cubic number of computer words upfront at initialization time –for the search stack contents and other structures (including copies of the distance matrix)– no additional memory is requested during the search, avoiding a slow-down to fragmented memory allocations.

We have also seen a possible approach to solve multi-dimensional packing problems by applying nested Monte-Carlo tree search with policy adaptation. The optimization algorithm is based on a combination of random choice and reinforcement learning and yields a trade-off between state space exploitation and exploration.

The implementation is flexible: It can handle additional placing constraints as well as alternative orientation. The obtained solution quality in a series of benchmarks is promising and calls for further refinements. One core advantage of the search is the anytime behavior: after executing the first random rollout in the NRPA a feasible packing is known. Another important feature are the low memory requirements. Only the amount of space for storing the packing and the policies at each level of the search and all container cells have to be present. As the rollouts can be executed in parallel, implementations on a multi-core CPU and a many-core GPU are possible. So far, we have worked with a single-threaded implementation, but experimented with several processes executed in parallel.

There are many interesting problem variants to be solved in container packing. In some cases. ISO-oriented boxes and the restriction to six possible orientations might be too inflexible, so that different angles and placements based on a CAD model of the object are of interest. As complete enumeration approaches are harder for this case, we expect further potential of applying the above randomized search approach.

Especially the inclusion of dynamics is crucial. If only parts of the conveyor belt are accessible, we have an online instead of an off-line optimization problem for which an algorithm with a good competitive ratio has to be designed. Additional to the packing there might also be ordering constraints, due to the partial delivery of products to the customers. If customer A is visited before customer B on a delivery tour, the objects should be placed in a way that it should be possible to unpack objects for A without moving packages for B.

In general, the logistic process chain of handling consumer goods in a distribution center is to unload them from a carrier to sort, sometimes store and finally distribute them. Transportation is the linking process of these steps. Carriers of the arriving goods are commonly pallets or, if the goods are loaded loosely, containers or trucks. There is a variety of solutions for the automation of most of the tasks mentioned above, except the unloading process. The manual execution of this process is a very tiring and not ergonomical activity, because there are many recurring movements and manipulation of goods with high weight.

21.6 Bibliographic Notes

Depth-first branch-and-bound (DFBnB) has been refined for the TSP [671]. *Symmetric TSPs* (STSPs), for which edge costs are the same in both directions, are usually optimally solved with algorithms relying on the quality of the Held-Karp lower bound [369]. Karp-Steel patching goes back to [383].

Agent-based commercial systems are used within the planning and controlling processes of containerized freight [164]. Team formation and interaction protocols have been designed for efficient resource allocation [564]. Agent-based systems have optimized planning and controlling processes within dynamic environments [257]. Ranges of application have been provided for industrial logistic processes [579].

For the TSP with release and due *time window constraints* (TSPTW), exact dynamic-programming algorithms exploit elimination tests to reduce the state space [173]. Variants can also be applied to TSPTW problems with precedence constraints. Frequently, branch-and-cut algorithms are the method of choice [24]. Introducing capacities constraints links to constraint solving.

In a study about the contents of containers arriving in European ports it was identified that 46.7% of the goods come in boxes of different size [177]. Another study [541] stated that for contract logistics the biggest number of parcels have a weight of 5-15 kg. On average 1,200 parcels fit in a 40'-container. Regarding this value the maximum payload of a 40'-container of approx. 30 tons can hardly be exploited by parcels. From the economical point of view for the full capacity of a container an optimal stacking is required with no or minimal gaps between the goods.

Nested Monte-Carlo (NMC) [105] is a randomized search method that has been successfully applied to solve many challenging combinatorial problems, including Klondike Solitaire [53]. NMC has also been applied to routing problems. For example, NRPA search has also been applied to efficiently solve the well-known Traveling Salesman Problem with Time Windows (TSPTW) optimally or very close to the optimum for small problem

instances with up to 50 cities [108]. Algorithmic refinements lead to solving single-vehicle pickup and delivery problems with time windows and capacity constraints with up to 200 cities to be visited [195].

There are several heuristic approaches to solve variants of the container loading problems [52, 166, 474, 406, 289], but they often have difficulties to scale to a larger number of objects, and do not cover orientation.

2D rectangle intersection for n objects can be tested by a divide-and-conquer approach in $O(n \lg n)$ time [312]. The drawback is that intersections are detected only after all rectangles are placed leading to many invalid placements during the rollout. In 3D the axis-aligned bounding boxes (AABB) algorithm is practically fast but requires $O(n^2)$ for the test. There is also the option of using range trees (with fractional cascading) for a query time of $O(\lg^2 n)$; the time for construction the range tree, however, rises to $O(n \lg^2 n)$ and is not incremental. Moreover, the algorithm is involved [151].

Rectangle packing has been studied in [416, 417]. Alternative approaches for non-square 2D packing problem have been addressed by [352] and high precision rectangles have been considered by [479, 353].

The work by Lim and Ying [452] proposes a new method for the 3D container packing problem that deviates from the traditional approach of wall building and layering. It uses the concept of *building growing* from multiple sides of the container. The idea of this 3D packing algorithm comes from the process of constructing a building. Boxes are placed on the wall of the container first as it builds the basement on the ground. After that, other boxes will be placed on top of the basement boxes. Following this process, boxes will be placed one on top of another. Every wall of the container can be treated as the ground for boxes to stack on. One drawback of the approach is that it does not consider gravity constraints.

Crainic et al. introduce the extreme point concept and present a new extreme point-based rule for packing items inside a 3D container [139]. The extreme point rule is addressed independently of the packing problem and can handle additional constraints, such as fixing the position of the items. The extreme point rule is also used to derive new constructive heuristics for the three-dimensional bin-packing problem. This rule was used in the space defragmentation heuristic [673].

Chapter 22
Additive Manufacturing

This chapter considers solving a problem in combinatorial search: the automated arrangement of *irregular-shaped* objects for industrial 3D printing. The input is a set of triangulated models; the output is a set of location and orientation vectors for the objects. The algorithm consists of three stages: 1) translation of the models into an octree; 2) design of an efficient test for pairwise intersection based on sphere trees; and 3) computation of an optimized placement of the objects using simulated annealing. We compare several sphere-tree construction methods and annealing parameter settings to derive valid packings.

22.1 Introduction

Additive[1] manufacturing (AM) has an increasing range of applicability. Compared to classical manufacturing it shows several advantages. Previously impossible shapes and structures are available, leading to prototypes that can be produced without a large supply or production chain. Hence, the manufacturing of new products is accelerated, the according costs are reduced, and a wide range of user-specified products can be produced. Given the 3D model of the product, it can be produced overnight and delivered to the consumers. While 3D printing (3DP) is one AM technique (processes that sequentially deposit material onto a powder bed with inkjet printer heads), nowadays, both terms used as umbrella terms for several technologies, which include laser stereolithography (SL), selective laser sintering/melting (SLS/SLM), electron beam melting (EBM), layer laminate manufacturing (LLN), and fused layer modeling (FLM).

To save production time and cost, the joint print of several objects is crucial. A valid *packing* for a set of objects $(o_1,\ldots,o_n)$ into a box $B=[0..x,0..y,0..z]$ with $(x,y,z)\in\mathbb{R}^3$ subject to objective function f is a sequence of location coordinates $(x_i,y_i,z_i)\in\mathbb{R}^3$ (e.g., for the centers of mass of o_i) and rotation angles $(\alpha_i,\beta_i,\gamma_i)\in[0,2\pi)^3$, $1\le i\le n$, which is collision-free (for all $1\le i\neq j\le n$ objects o_i and o_j do not overlap), fits completely in box B, and optimizes f. Packing algorithms should: 1) be robust to overcome inaccuracies in the input model, 2) support general user-supplied objective functions, 3) preserve a minimal pairwise distance between the objects.

22.2 Sphere-Tree Construction

Bounding volume hierarchies (BVHs) are recursive tree data structures that at the leaf nodes include a primitive volume data type. There are various sorts of BVHs, e.g., based on AABBs, OBBs (oriented bounded boxes), cones, ellipsoids, and convex hulls. With *sphere trees* (Figure 22.1) we chose BVHs for which translation,

[1] This chapter is based on joint work with Paul Wichern. It improves the work from [234].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_22

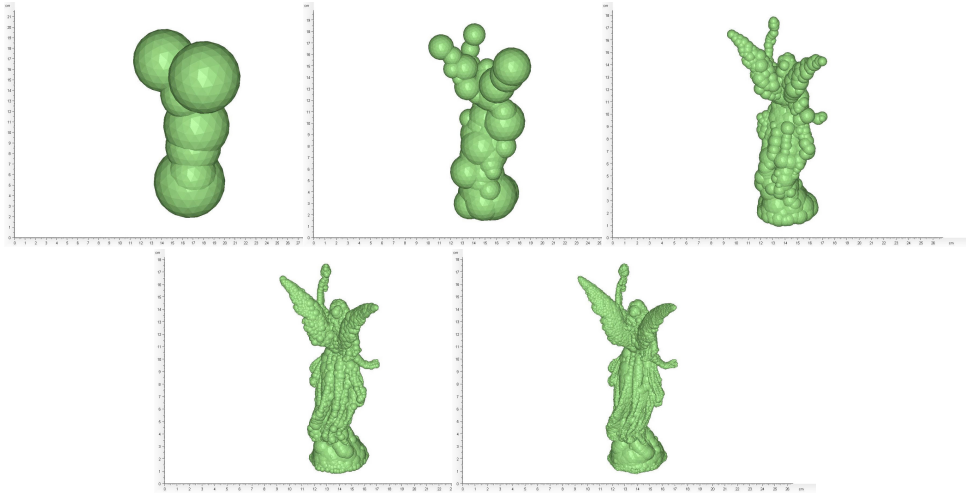


Figure 22.1: Levels of a sphere tree.

Algorithm 22.1: Creation of an Octree.

procedure InOutOctree(T, b_T, c_{min})
$O \leftarrow$ octree of T with smallest cube around b_T as root
for $c \in$ cells of O **do**
  mark c as INTERNAL
$c_{start} \leftarrow$ corner leaf cell of O
BreadthFirstMark(O, c_{start}, EXTERNAL)
return O

rotation and intersection are fast: a collision of spheres A and B wrt origins c_A and c_B and radii r_A and r_B is detected by evaluating $\|c_A - c_B\|_2 < r_A + r_B$. If there is no intersection on a coarser level of granularity higher up in the tree, there will be none on a finer level.

The midpoint (x,y,z) of the sphere on four points (x_i,y_i,z_i), $1 \leq i \leq 4$, together with the check of collinearity, can be computed by evaluating the equation

$$det \begin{vmatrix} x^2+y^2+z^2 & x & y & z & 1 \\ x_1^2+y_1^2+z_1^2 & x_1 & y_1 & z_1 & 1 \\ x_2^2+y_2^2+z_2^2 & x_2 & y_2 & z_2 & 1 \\ x_3^2+y_3^2+z_3^2 & x_3 & y_3 & z_3 & 1 \\ x_4^2+y_4^2+z_4^2 & x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0$$

Octree The simplest algorithm to construct a sphere tree is by extending the corresponding octree. Algorithm 22.1 distinguishes in- and outside leaf cells in the octree, with T as the set of object triangles and b_T as its bounding box, and c_{min} as minimal cell size. It marks cells in breadth-first order, starting with an initial external one. All cells belong to the interior of the object. For the subsequent construction of a sphere tree, in Algorithm 22.2 a sphere is generated for each internal cell (see Figure 22.2).

Medial Axis Another sphere-tree construction method exploits the *medial axis*, a generalized Voronoi nearest-boundary distance diagram data structure residing in the interior of objects (see Figure 22.3). As the exact medial axis can be complex, computational approaches for its construction usually resort to its approximation.

Sphere-tree construction via the medial axis operates in stages (illustrated in Figure 22.4). In the first stage, an initial 3D Voronoi diagram *for the object vertices* is constructed that approximates the exact medial axis. The Voronoi edges/faces that do not cross the object boundaries are the building blocks of the medial axis skeleton.

Algorithm 22.2: Sphere construction via Octree.

procedure SphereByOctree(T, b_T, c_{min})
$O \leftarrow InOutOctree(T, b_T, c_{min})$
$S_{tree} \leftarrow O \setminus \{o \in O \mid o \text{ marked as EXTERNAL}\}$
replace every cube in S_{tree} by its surrounding sphere
return S_{tree}

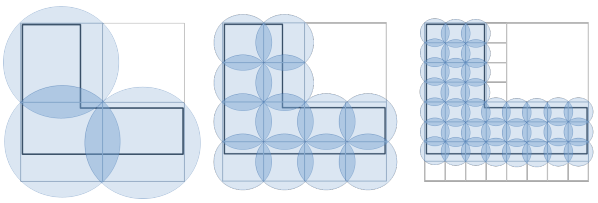


Figure 22.2: Conversion of an octree into a sphere tree.

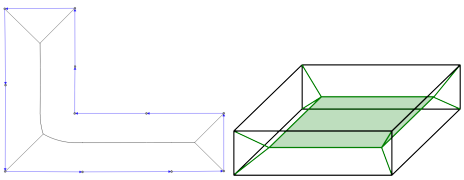


Figure 22.3: Exact medial axes in 2D and 3D.

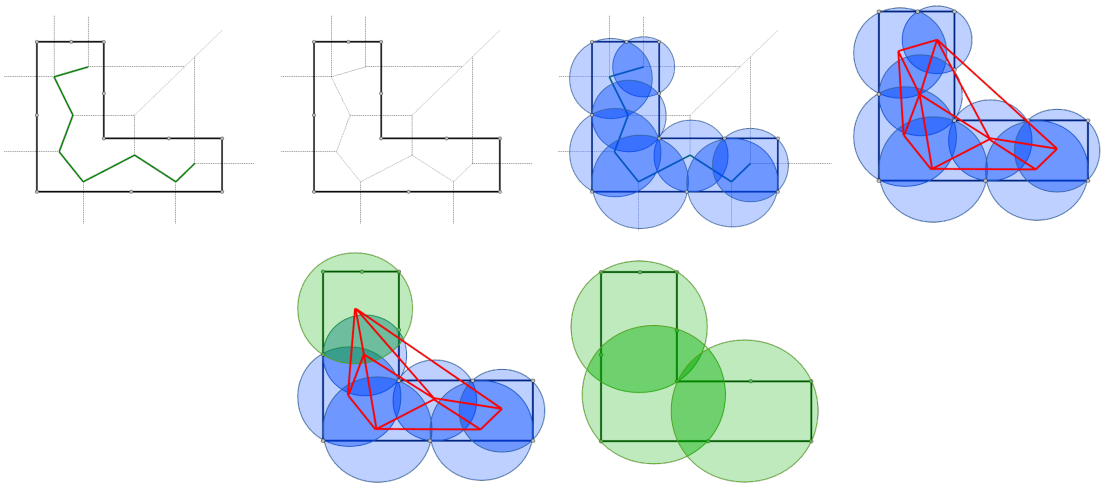


Figure 22.4: Constructing a sphere tree via medial axis approximation: 1) computing a Voronoi diagram of the object vertices, 2) extracting the approximation of the medial axis of the object, 3) generating a sphere cover, 4) computing the triangulation of centers, 5) merging spheres.

Together with the distance to the boundary they define the set of spheres to cover the object. Then, with the help of a triangulation of the centers (red), we incrementally merge spheres that have been constructed. Too aggressive merging strategies, however, negatively influence the runtime and are avoided.

For complex objects (see Figure22.5), the Voronoi diagram is *extended by sampling random points on the object surface*. Such *adaptive sampling* starts with a set of points on the surface of the object, and is extracted from the medial axis approximation. One sampling option is to exploit an underlying grid, but on curved boundaries

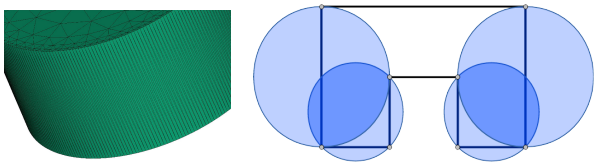


Figure 22.5: Curved object (left), and one, where all vertices, but not all edges are covered (right).

with small but lengthy triangles, cells might still not be sampled. Therefore, we decided to recursively construct coverage points (see Figure 22.6).

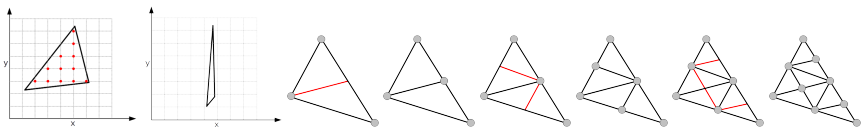


Figure 22.6: Problem of grid sampling (left) and recursive construction of coverage points (right).

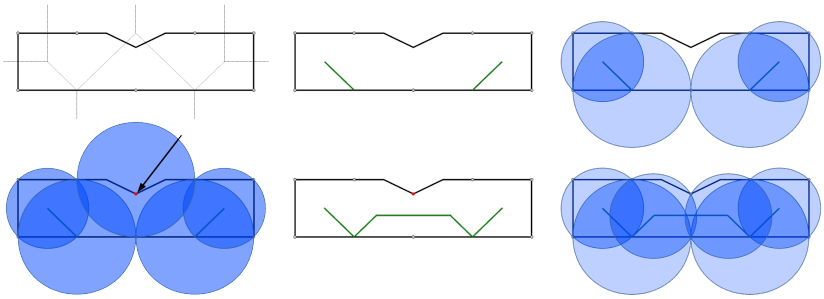


Figure 22.7: Steps in *ImprovedAdaptiveSampling*: 1) Voronoi diagram, 2) medial axis approximation, 3) spheres constructed, 4) new point generated (red), 5) updated axis, 6) final sphere cover.

We apply coarse sampling for the initial Voronoi diagram and insert the additional set of *coverage* points on the surface for the resulting spheres (Figure 22.7) in a refinement step. Followed by this, an error value is computed, which denotes how far a sphere exceeds the surface. For all spheres that have a value that is too large, the medial axis is refined, until no sphere remaining in the result set exceeds the error threshold. Eventually, all sample points and the model itself were finally covered with spheres.

Sphere Packing The sphere packing algorithm operates in voxel space, which is a discretized grid representation of the workspace. The input is the set of octree cells that have been identified as being inside. The algorithm incrementally adds a sphere in the cell that has the largest distance to the surface. Next, all distances are updated, and the algorithm iterates (see Figure 22.8). As the result of the algorithm needs to be an object, spheres must be inflated to provide a complete object cover.

22.3 Robustness Considerations

Figure 22.9 illustrates subtle but important aspects that affect the robustness of the packing algorithms, namely orientation, holes, and self-intersections. Other issues are isolated triangles, open and multiple edges, or degenerate triangles.

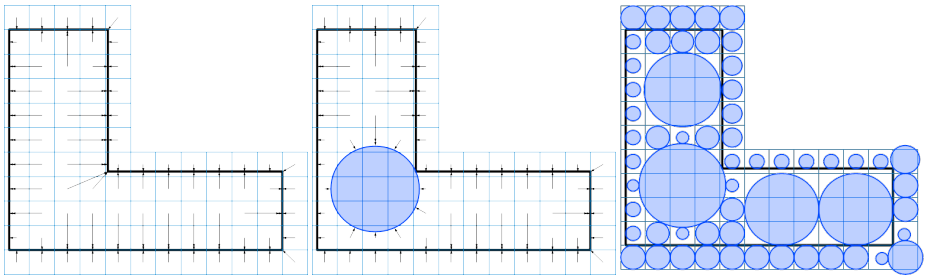


Figure 22.8: Steps in the construction of a sphere packing (top to bottom, left to right): 1) determining distances of grid centers to the nearest boundary point, 2) drawing a corresponding sphere, updating the distance information, 3) final result of iterating the process

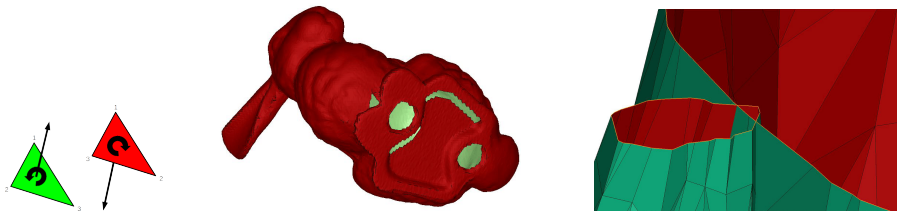


Figure 22.9: Robustness issues: wrong orientation & holes (left), self-intersection (right).

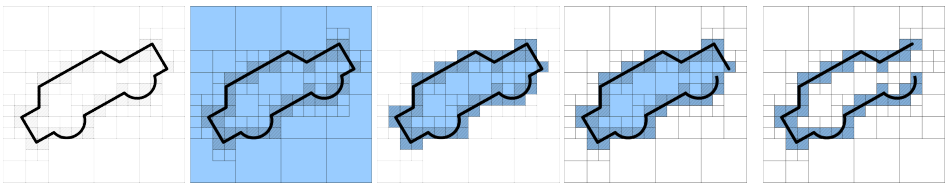


Figure 22.10: Separating in- from exterior octree cells (top), impact of small and large holes (bottom).

Objects may have inverse orientation, defined by the normals of the triangles. As an example, in the visualization the model *bunny* (left) is shown in red, illustrating that its orientation is reversed. The algorithm is robust with respect to this artifact.

For interior detection Figure 22.10 shows that small holes may be captured correctly, but for larger holes there might be semantic problems that arise. The approach of imposing a minimal cell size for an object boundary aligns with the observation that a 3D printer assumes that even a flat surface has some positive volume.

the runtime, so that good trade-offs between running time and robustness are needed. For illustration purpose, we conducted an experiment for sphere computation with very large coordinates that with data type *float* (IEEE Standard 754) took 0.11 seconds and produced 50,006 errors, with data type *double* took 0.12 seconds and produced 481 errors, with data type *SoftFloat128* we got 1.21 seconds and six errors, and with *BigIntegers/Rationals* 51.53s and no error. We used a combination of the data types with a quick check of validity that achieves the acceptable trade-off with 0.43 seconds, while still producing no error.

22.4 Global Optimization

To evaluate a state in Algorithm 22.3 we assume a set of models M, their sphere trees S_{trees}, a build envelope e, a set of evaluation functions $F = \{f_1, \ldots, f_n\}$, and a score of colliding leaves δ.

Algorithm 22.3: State evaluation function.

procedure EvaluateState(M, S_{trees}, e, F, δ)
$c_{collisions} \leftarrow 0$
for $s_{tree} \in S_{trees}$ **do**
  $c_{collisions} \leftarrow c_{collisions}$ + number of leaves in s_{tree} not completely inside e
for $a_{tree}, b_{tree} \in S_{trees}, a_{tree} \neq b_{tree} \wedge$ index of $a_{tree} <$ index of b_{tree} **do**
  $c_{collisions} \leftarrow c_{collisions} + CollisionCount(a_{tree}, b_{tree})$
return $\delta \cdot c_{collisions} + \sum_{i=1}^{n} f_i(M)$

Algorithm 22.4: Estimating the amount of overlap.

procedure *CollisionCount*(a, b).
if a and b do not overlap **then**
  return 0
else if a and b are leaves **then**
  return 1
else
  $c \leftarrow 0$
  for $a_{child} \in$ children of a **do**
    for $b_{child} \in$ children of b **do**
      $c \leftarrow c + CollisionCount(a_{child}, b_{child})$
  return c

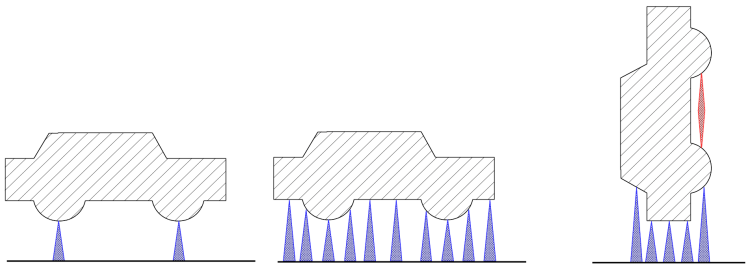


Figure 22.11: 3D printing support triangles of different granularity and object orientation.

Algorithm 22.4 recursively computes the number of intersections of spheres tree leaves, which is combined with the overall objective function (such as maximizing centrality and minimizing height). For more advanced optimization functions, the user can supply his own evaluation function. For example, additional support (see Figure 22.11) is dependent on the rotation of the object and needed not only for stabilization of the object but also for the transport of heat. Its cost must be implemented in a user-defined objective function.

Given the neighborhood relation and the evaluation function, an initial packing can be optimized. We choose simulated annealing (SA, see Algorithm 22.5) as the global optimization process. It allows sub-optimal decisions with a probability that is decreasing with the temperature *temp*. Differently from the research of finding an optimal 3D AABB packing, we used simulated annealing with respect to minimal translation step size $q_{stepsize}$ and rotation stepsize q_{angle}. Using sphere trees, the search primitives for checking intersection and translation are fast, so that SA converges more effectively to a good solution.

Complexity Considerations Collision counting has a worst-case time complexity of $O(n_1 n_2)$ for two trees of size n_1 and n_2 but is faster in practice.

Assuming m to be the number of models, $|T|$ to be the number of input triangles, n to be the number of cells in the octree, c to be the number of cooling steps, n_1, n_2 to be the number of nodes in two sphere trees, and n_{max} to be the number of nodes in the largest sphere tree, we get the worst-case run-time complexities shown

Algorithm 22.5: Packing optimization via simulated annealing.

```
procedure SimulatedAnnealing(s, f, c)
i ← 0
u ← u_best ← s
temp ← 1
while temp > 0 do
  u_next ← ExpandRandom(u, temp)
  if f(u_next) < f(u) then
    u ← u_next
    if f(u) < f(u_best) then
      u_best ← u
  else
    r ← Random(0, 1)
    if r < e^((f(u_next)−f(u))/temp) then
      u ← u_next
  i ← i + 1
  temp ← Cooling(temp, i, c)
return u_best
```

Table 22.1: Complexity of SA, assuming octree construction.

Algorithm	**Run-Time Complexity**
InOutOctree	$O(\|T\|)$
SphereTreeByOctree	$O(\|T\|)$
CollisionCount	$O(n_1 n_2)$
EvaluateState	$O(m^2 n_{max}^2)$
SimulatedAnnealing	$O(cm^2 n_{max}^2)$
Total	$O(m\|T\| + cm^2 n_{max}^2)$

in Table 22.1. For the sake of brevity, we assume sphere trees generated directly from the octrees. Finding the exterior cells may take $O(|T|)$ time, which dominates sphere-tree construction.

SA terminates if the temperature after c steps has reached zero. The running time of one step heavily depends on *EvaluateState*, which is supplied by the user and, therefore, can be arbitrarily complex. In Algorithm 22.3 we chose a default implementation for the evaluation function based on collision counting, for which *EvaluateState* has a worst-case bound of $O(m^2 n_{max}^2)$ so that the overall time complexity is $O\left(m|T|n + cm^2 n_{max}^2\right)$ (see Table 22.1).

22.5 Experimental Results

We take publicly available 3D CAD models, including the model *cow* of the University of North Carolina (with 5,804 triangles), the model *bunny* of Stanford University (69,451), and the model *ShowPart* from Renishaw (250,934, see Figure22.12). Further models are *angel*, *dragon*, *hand*, *buddha* and belong to the Stanford 3D Scanning Repository and Greg Turk's Large Geometric Models Archive.

Table 22.2 compares some selected sphere tree construction algorithms. For the sphere tree computed via medial axis, we also measure the impact of a larger branching factor of interior nodes. A visualization for applying three different construction algorithms to the *cow* model is shown in Figure 22.13.

For the initial state we placed the objects on a sphere around the center and started the simulated annealing process (maximizing centrality or minimizing height, see Figure 22.14). As expected, the octree construction

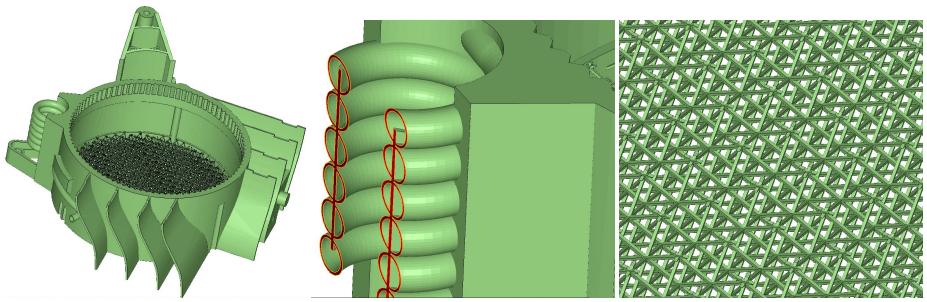


Figure 22.12: Complex 3D object with magnified parts.

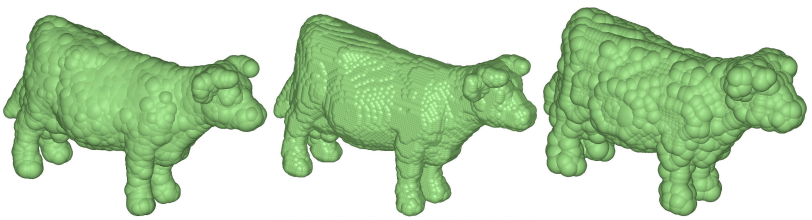


Figure 22.13: Sphere trees constructed via medial axes, octrees, and sphere packings.

Table 22.2: Sphere-tree construction (b: branching factor).

Model	Approach	Leaves	Depth	Time
ShowPart	Octree	105,027	8	0.95s
	Medial Axis ($b = 16$)	33,483	6	44.27s
	Medial Axis ($b = 8$)	33,483	8	45.64s
	Medial Axis ($b = 4$)	33,483	11	53.45s
	Medial Axis ($b = 2$)	33,483	24	78.58s
Bunny	Octree	14,095	7	0.20s
	Medial Axis ($b = 16$)	25,755	6	23.86s
	Medial Axis ($b = 8$)	25,755	8	25.07s
	Medial Axis ($b = 4$)	25,755	11	27.14s
	Medial Axis ($b = 2$)	25,755	21	36.91s
Cow	Octree	8,142	7	0.03s
	Medial Axis ($b = 16$)	1,634	4	1.70s
	Medial Axis ($b = 8$)	1,634	6	1.76s
	Medial Axis ($b = 4$)	1,634	8	1.80s
	Medial Axis ($b = 2$)	1,634	17	2.10s
Angel	Octree	28,790	8	1.44s
	Medial Axis ($b = 16$)	7,875	5	14.75s
	Medial Axis ($b = 8$)	7,875	7	15.24s
	Medial Axis ($b = 4$)	7,875	10	15.98s
	Medial Axis ($b = 2$)	7,875	20	18.57s
Dragon	Octree	14,394	7	2.06s
	Medial Axis ($b = 16$)	13,961	6	23.64s
	Medial Axis ($b = 8$)	13,961	7	24.24s
	Medial Axis ($b = 4$)	13,961	11	25.10s
	Medial Axis ($b = 2$)	13,961	21	31.58s
Hand	Octree	5,390	7	1.38s
	Medial Axis ($b = 16$)	3,866	5	12.54s
	Medial Axis ($b = 8$)	3,866	6	12.99s
	Medial Axis ($b = 4$)	3,866	9	13.23s
	Medial Axis ($b = 2$)	3,866	18	14.23s
Buddha	Octree	48,428	8	3.00s
	Medial Axis ($b = 16$)	15,080	5	32.31s
	Medial Axis ($b = 8$)	15,080	7	33.65s
	Medial Axis ($b = 4$)	15,080	11	34.24s
	Medial Axis ($b = 2$)	15,080	22	44.22s

was by far faster, but in all but one case (bunny) the medial axis yielded fewer leaf nodes of the sphere tree. Decreasing the branching factor further slows down sphere-tree construction.

Besides changing the user-supplied optimization function we experimented with different branching strategies in the simulated annealing algorithm: translation and/or rotation on one/all axis and of one/all models; the best results were obtained with translating or rotation of one model and one axis at a time. Dynamically adapting the cooling to the number of rotational and translational steps was fortunate (see Figure 22.16).

Despite the considerably larger number of leaves, the intersection test for sphere trees based on the octree data structure was the fastest. As indicated in Figure 22.15 with tight upper and lower bounds for the cover, a

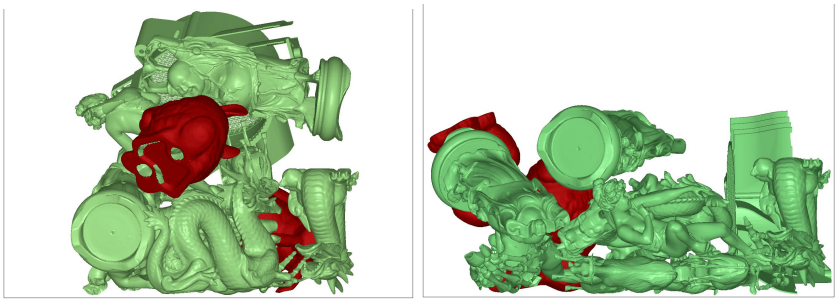


Figure 22.14: Arrangement of 3D objects, maximizing centrality (left) and total height (right).

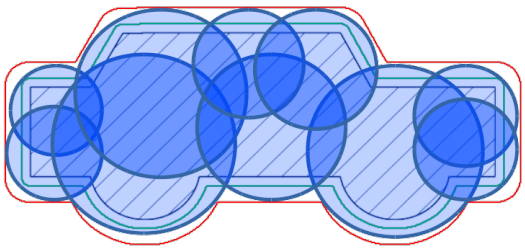


Figure 22.15: Min. (blue) and max. distance (red) of medial axis sphere-tree cover of object (green).

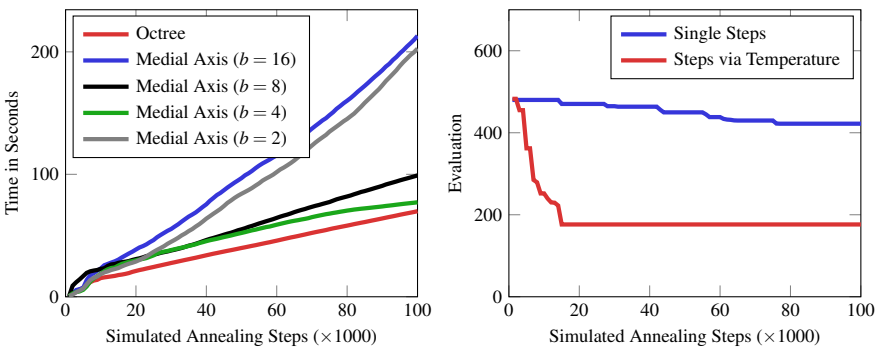


Figure 22.16: Result of the optimization process.

possible reason for this unexpected behavior might be that the spheres in the octree have a smaller overlap than for the construction via the medial axis. Moreover, in the optimization algorithm we do not aim at the Boolean decisions, but compute the number of intersecting spheres. For sphere trees constructed via the medial axis computation, we obtain the best performance with a sweet spot branching factor $b \approx 4$.

22.6 Summary

In the area of combinatorial search there is a large body of research on packing regular-shaped objects (squares, rectangles, boxes). In this chapter we have seen an approach to solve the packing problem of irregular-shaped objects, which has practical implications on 3D printing to save both production time and cost. The goal was

to find a collision-free arrangement preserving a minimal distance between the objects and optimizing a user-defined objective function.

The algorithm is practical and its refined implementation is used in industrial practice, which has direct implications on the software's flexibility and quality. The user can put additional requirements in the objective function, like a small height, a high surface quality, a low number of supports, less stretch, and small distances for the laser travel. Advanced topics are stability and extractability. For the concurrent print we allow flexible change to the objective function as the evaluation changes quickly with respect to customer demands, varying hardware, and chosen materials.

The research interest is studying the data structure for engineering the efficiency of the optimization algorithm, which boils down to frequently computing some score for the intersection of objects. The experimental study reveals that there might not be a uniformly best intersection routine. While medial axis and sphere packings produce a smaller number of spheres, computing the intersection volume is often slower than using sphere trees extracted straight from the octrees.

The packings we found at the end of the simulated annealing process were all valid. The results were better if the step size was dynamically adjusted to the temperature parameter. The simple evaluation function relied on counting the number of intersections (as an indicator for computing the volume of the sphere tree intersection area) to improve the arrangement of objects. Other global optimization procedures only need efficient intersection tests for reducing the set of possible successor candidates.

22.7 Bibliographic Notes

A number of research papers on the efficient packing of objects for 3D printing have been published: [660] separates the work into two classes: 3D packing and searching for an optimal orientation, which we consider in common; [360] applies genetic algorithms to place the models close to the working space center using a hierarchical structure of axis-aligned bounding boxes (AABBs) for collision detection; [492] improves manufacturing time, surface quality and the volume of support also using genetic algorithms for the optimization; [500] optimizes average surface quality and manufacturing time, comparing particle swarm optimization with genetic algorithms for finding the Pareto optimum; [140] provides an overview on AABB algorithms, while [237] is concerned about heuristically packing concave/convex bodies, assuming no noise in the input; [660] optimizes height, surface quality and support volume, as well as genetic algorithms and octrees for AABB collision detection, thus being limited to rotations of 90 degrees. Another option is to sample random packings, while incrementally learning their best arrangement [197].

Bretshaw [78] calls the algorithm of [277]. Depending on the density of the sampling [355], the approximation can be made arbitrarily exact. 3D Voronoi diagrams are the geometric dual of the according Delaunay tessellations, which are more convenient to compute [646]; we used the algorithm of Bowyer and Watson [313]. For querying points in such a nearest-neighbor database, randomized data structures and random walk algorithms are recommended [485, 156].

Packing squares into rectangles has been studied in [416, 417], discrete rectangles in [352] and high-precision rectangles in [479, 353]. A new method for packing boxes grows an arrangement of objects from multiple sides of the container [452]. Industrial tools with restricted functionality for packing irregular-shaped object in the context of 3D printing include Magics Sintermodule and NetFab Professional. Other tools (CAMWorks, MOSAIX, Nest++, ProNest, Nshaker& NEstimate) are limited to 2D.

Chapter 23
Robot Motion Planning

Motion planning aims to increase the ability of robots to plan and act on their own. Efficient and safe motion planning algorithms are crucial for the applications of robots in life and industry. One of the most developed motion planning for these requirements is sampling-based motion planning, such as rapidly exploring random trees (RRT), and its improvement RRT*.

We will look at the *physical traveling salesman problem* (PTSP), i.e., the simulation of the continuous operation of a vehicle for finding multiple goals in a gridworld with obstacles. After we precompute cell-precise single-source shortest paths, we determine a tour to follow, solving different variants of the TSP. For moderately-sized problems, we apply optimal depth-first branch-and-bound with $O(1)$ time spent per search tree node. For larger problems, we apply randomized search with policy adaptation to learn from good tours.

Next, we develop an efficient approach that enables a more realistic robotic vehicle to inspect all the regions of interest while avoiding collisions and reducing the distance traveled. Again, a key aspect of the approach is the transformation of the multi-region inspection task into a clustered TSP. This is achieved by generating several inspection points on the medial axis of each region to increase visibility and by grouping the inspection points according to their color. We implement a solver based on branch-and-bound search to effectively find low-cost tours. These tours serve to guide sampling-based motion planning as it expands a motion tree in search of a collision-free and dynamically feasible trajectory to carry out the multi-region inspection task. Sampling-based motion planning and the TSP solver work in tandem, exchanging information to improve the quality of the tours and motion trajectories. The approach is evaluated in simulation, using high-dimensional models with second-order dynamics of robotic vehicles to carry out inspection tasks in complex environments. For surface inspections we are given a 3D environment with a set of objects that need to be inspected, and an inspection quality α. The objective is to compute a set of waypoints whose joint visibility ratio is at least α and a dynamically feasible and collision-free trajectory that enables the aerial vehicle to reach all the waypoints. The approach minimizes the number of the waypoints and the overall distance traveled by the aerial vehicle.

23.1 Introduction

As[1] a combination of task and motion planning, we are concerned about motion planning for multiple goals.

Additionally, to the classical TSP that calls for minimizing the total distance traveled within a tour of the salesman, the *physical traveling salesman problem* (PTSP) features a model of velocity that leads to changes of system states in matters of direction, friction and acceleration of the agent's vehicle. The objective of the PTSP is to visit the maximum number of waypoints of the map in the minimum number of time steps. The map of the

[1] This chapter is based on joint work with Xuzhe Dang, Mihai Pomarlan, Erion Plaku, Baris Can Secim, Sara Rashidian, and Yazz Warsame. It puts together and improves the work work from [145, 216, 222, 645, 527].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_23

environment can be flexible and may resort to a two-dimensional board, where waypoints are scattered around, and multiple obstacles are present.

While the single-objective PTSP only calls for minimizing total simulation time, the so-called multiobjective PTSP (MO-PTSP) additionally calls for minimizing other criteria, e.g., the fuel level and damage to the vehicle. Hence, a variety of other objectives which affect these criteria have to be taken into account.

Given that TSP is NP-hard (even in the strong sense), with a rising number of waypoints, real-time constraints and complex side-effects, PTSP is a computational challenge. We apply *policy adaptation* for computing tours and for controlling the vehicle; integration of induced physics, like angular or surface change, and more general cost functions; and *graph search engineering*, i.e., the adaptation and tuning of the shortest paths search algorithms (SSSP), resulting in high-speed graph search.

We generalize the problem to colored waypoints, where from each color at least one waypoint has to be collected. This variant is called *generalized TSP*, GTSP for short. The *depth-first branch-and-bound* (G)TSP solver is optimal. We study *complexity results for solving SSSPs and (G)TSPs*: the backtrack solver requires at most constant time per node and SSSP has a linear-time worst-case overall performance.

As defects to objects such as pipeline leakage can result in tremendous economic loss, *surface inspection* is one of the most important problems in modern robotics. Given a set of objects located in an environment with obstacles, the task is to find a least-cost obstacle-avoiding path that inspects all objects.

A common approach to solve the inspection problem in polygonal environments is to first compute a minimal set of waypoints by solving the art gallery problem and then to use an efficient TSP solver to compute a path for the robot towards visiting these waypoints. Both subproblems are NP-hard, so that the combined result might only be approximate for the entire task. Moreover, such solution waypoints are often attached to walls and obstacles and, thus, difficult to reach.

We use colors to ensure that regions with the same color are inspected in order. Many approaches that consider visibility and obstacle-avoidance constraints on the tour do not provide a mechanism to specify constraints on the order in which the regions are inspected. In a first setting, the environment, obstacles, and regions of interest are represented as a bitmap image. A crucial aspect of the presented approach is the transformation of the inspection problem into a *clustered* TSP (CTSP) over an undirected, colored, and weighted graph. This is achieved by first generating waypoints that will enable the robotic vehicle to inspect each user-specified region of interest.

In 2D, we adapt a grass-firing transformation algorithm and subsequent filtering to effectively generate these waypoints on the medial axis of each region and to ensure that the entire region can be inspected by visiting its waypoints. In 3D, we use skeletonization and waypoint sampling together with filtering. By grouping the waypoints according to their regions, the inspection problem is transformed into a CTSP where each point inherits the color of its region. A depth-first branch-and-bound search efficiently solves the resulting CTSP and computes a low-cost tour that enables the robotic vehicle to inspect all the regions. Finally, a controller is employed to drive the robotic vehicle from one waypoint to the next as specified by the tour.

The efficiency of the approach derives from its ability to generate a small number of waypoints while at the same time obtaining a good coverage of the regions that need to be inspected. The approach is evaluated in sampling-based motion planning within a simulation using high-dimensional models with second-order dynamics of robotic vehicles to carry out inspection tasks in complex environments.

23.2 RRT, RRT*, and Deep RRT*

The algorithm *rapidly exploring random tree* (RRT) expands a search tree by randomly generating new samples and connecting them. This sampling strategy is fast and efficient. It finds a collision-free path for the robot in a low-dimensional space. However, the number of samples generated by this random strategy will grow fast with

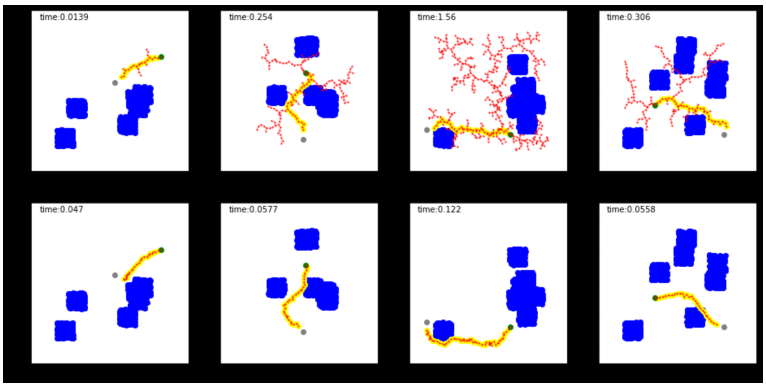


Figure 23.1: Comparison of RRT* and Deep RRT*. The top row shows the results of RRT*. The bottom row shows the results of Deep RRT*. The blocks points represent obstacles. The curly lines represent the tree nodes.

the increasing sizes and dimensions of the space, so that algorithms like RRT and its improved variant RRT* (refining the search tree by shortcuts) will be computationally expensive.

Algorithm	Success Rate	Tree Size	Time
RRT*	97.4%	199.87 ± 150.01	0.60 ± 0.72
Deep RRT*	90.1%	74.28 ± 126.93	0.36 ± 1.19

Table 23.1: Resuts RRT* vs Deep RRT*

Instead of using a random sample strategy, Deep RRT*, however, trains a model to *guide* the sampling strategy. The model uses a similar architecture to MPNet that contains two neural networks – the encoder network and the policy network. The first part of the model, the encoder network, embeds the obstacles state space that is represented by point clouds into a latent space. The second neural network is a policy network (see Chapter 4). Given the current tree node state, the goal state, and the latent space from the encoder, the policy network predicts the distribution of the subsequent node. Then, Deep RRT* expands the tree by sampling a new node with the predicted distribution. The model, both encoder network and policy network, is trained by self-play. In each iteration, Deep RRT* finds N successful paths from trees with a set of predefined state spaces, randomly generated initial states, and randomly generated goal state, and then store these paths in a rollout buffer in each epoch. Then, Deep RRT* trains the model.

RRT* was implemented as benchmark algorithm and compared with Deep RRT*. We trained and tested Deep RRT* in the same 2D environments used by MPNet. The 2D environment contains 110 different scenarios. We used 100 scenarios to train the model and keep the remaining scenarios for testing. Each testing scenario contains 2000 different initial and goal configurations. We randomly selected 500 configurations of those. In the experiments, the maximum number of nodes of the tree is set to 1000 for RRT*, and 300 for Deep RRT*. During the planning, once a node expanded is located in the goal region, the path is returned. Otherwise, the problem is considered unsolved.

Figure 23.1 shows four scenarios, each solved by RRT* (top row) and Deep RRT* (bottom row). Table 23.1 presents the success rate, search tree size and CPU-time comparison of Deep RRT* against RRT* on the testing scenarios. As Table 23.1 shows, Deep RRT* is more efficient and faster than RRT*. The size of the search tree built by Deep RRT* is only 37.16% of the size of the search tree built by RRT*. The average time to find a path with Deep RRT* is 0.36s, and the average time to find a pathwith RRT* is 0.6s. The success rate of Deep RRT* is, however, reduced by 7.3%. We find that in most of unsuccessful cases, when the search tree is expanded close to the goal area, the new node generated by Deep RRT* misses it and generates samples randomly in the area close to the goal until it reaches it, or the maximum number of samples is exceeded. Adding weighted entropy to the loss function and reducing it during the training helps.

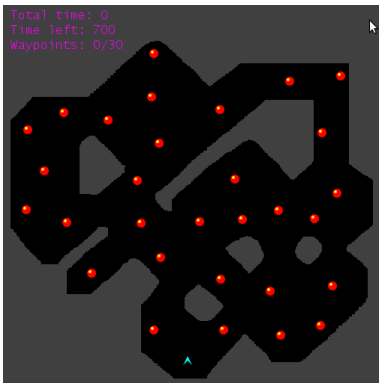


Figure 23.2: The Physical TSP.

23.3 Physical TSP

The main purpose of the PTSP (see Figure 23.2) is to provide a benchmark for combined task and motion planning in interactive computer games. Within the PTSP framework, ships move autonomously through the gridworld by applying thrust and rotation, with up to six different actions available to the ship (a Boolean input indicating the ship either to accelerate or not, and a number indicating a rotation to the left, to the right or no rotation at all). All actions are applied to the ship as forces that update its position, orientation, and velocity at each time step. The framework provides different sets of 2D benchmark maps of blocking and non-blocking cells with 10-50 waypoints. The ship can only move along non-blocking cells to reach the waypoints.

The crucial parts of the software framework are the *controller* and the *TSP solver*. The controller adapts the map to a proper world model and determines the shortest path from the current position of the ship towards every available waypoint. The solver then calculates a solution to the TSP based on the shortest paths provided by the controller and reports it back to the controller. Given the TSP solution, the controller navigates the ship by performing multistep operations (called *macro actions* or *macros*) which consist of acceleration, rotation, and braking.

In the default controller of the framework, each map is approximated by a weighted graph (each 64 cells are merged into one graph node), in which we compute pairwise shortest paths between the start location and each of the waypoints. These distances are then fed as a matrix into a TSP solver, and –utilizing the imposed schedule of waypoints– the controller software performs random search on macros. It is not required to return the ship to the location where it started from, which transforms the search for a (min-cost) simple cycle to a simple route in the underlying graph.

The execution model of the framework is real-time: actions have to be committed at a rate of about 40ms. The startup time is 0.1s for each waypoint in the scene. As part of the framework, besides computer play, the interface allows replay of preceding games, and human players to participate in solving the problem using interactive steering.

In one generalization of the PTSP (see Figure 23.3, left) we aim at solving a PTSP with colored waypoints. In a tour each color has to be visited at least once, so that all colors but not necessarily all waypoints are visited in a tour. This *generalized PTSP* (GPTSP) is a natural extension to the PTSP that is related to finding a *clustered PTSP* where we must visit every waypoint in a pre-defined cluster before approaching the next.

Another variant of the PTSP extends the above features by additional types of obstacles. This provides a testbed for multi-objective PTSP (MO-PTSP) solutions. While reaching all waypoints is still the primary objective, three different, equally important secondary objectives have been chosen: *time* taken to reach every waypoint, *damage* inflicted on the ship, and the amount of consumed *fuel*. Fuel can be picked up in form of fuel tanks. Additionally, the extended framework provides different types of obstacles as shown in Figure 23.3 (right): *lava* is a non-blocking cell which inflicts high damage on the ship; *damaging surface* is a blocking cell which inflicts

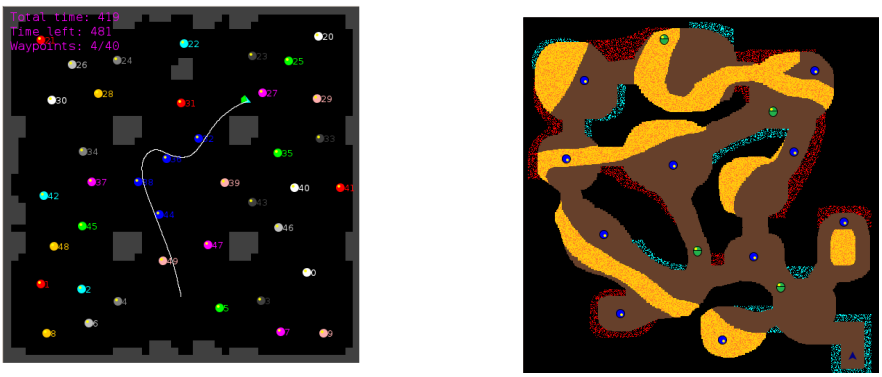


Figure 23.3: An instance for the Generalized PTSP with waypoint IDs (left), and one for the MO-PTSP (right).

high damage on the ship in case of collision; and *elastic surface* is a blocking cell which makes the ship bounce back without taking any damage in case of collision.

More formally, the *object state* consists of a triple $o = (c, v, d)$ of location $c = (c_x, c_y)$, velocity $v = (v_x, v_y)$ and direction $d = (d_x, d_y)$, which are changed according to the imposed dynamics of the system. An additional color may be associated to o.

Waypoints are static objects. There are other factors involved, like the size of the ship for collision detection, that we abstract from: for collision it is assumed to be $3/2$ times a predefined radius, so that by enlarging the obstacles by this radius (or by taking Minkowski sums) we assume that the location of the ship is a point in Euclidean space.

A *map* is a labeling of cells with object types (e.g., `@` for obstacle, `C` for waypoint, `L` for lava, `_` for freespace, `S` for ship). A *game state* g is a triple (o, F, W) of current object state of the ship together with a list F of fuel tanks and a list W of waypoints visited. The *score* of a game state is a triple $s = (t, d, f)$ and records the time t consumed so far, the damage d taken, and the fuel level f of the ship. It is possible to optimize the values t, d and f individually or to take a linear combination: with each set of coefficients $(\alpha_1, \alpha_2, \alpha_3)$ in optimizing the cost function $\alpha_1 \cdot t + \alpha_2 \cdot d + \alpha_3 \cdot f$ one can find an element of the so-called *Pareto frontier*.

For precomputing an order of waypoints, the PTSP is discretized to a non-physical TSP variant. In terms of object states, $o = (c, v, d)$ is projected to location $\phi(o) = c = (c_x, c_y)$. Even though this is a rough approximation, solving the standard TSP often gives a plausible ordering for the PTSP. Because of the physics involved, we model the problem as an Asymmetric TSP (ATSP), which in general is more difficult to solve, but in return more flexible for incorporating complex cost functions.

23.4 Shortest Paths

To find an appropriate schedule of waypoints, shortest paths in the octile gridworld are precomputed via projecting the current system state to the grid cell. In essence, by precomputing shortest paths, the gridworld is contracted to a weighted graph of waypoints.

The shortest paths from each waypoint to each pixel is precomputed and stored in an array for further lookups. One may apply *flood-fill* single-source shortest path (SSSP) search implementation that was studied in Chapter 2. While color filling assigns a color to each cell starting by the initial one, the SSSP variant is aimed at assigning shortest path distances to the cells. Instead of cost-first search, the *flood-fill* algorithm explores the gridworld row-wise. Cells (beyond an obstacle) in the rows above and below the current position are marked for further processing and put into a queue. After each extraction of a node from the queue, the process continues testing shortest-path improvement in each visited cell.

The result is an improved *locality* of the search; given that memory storage is row-major, processing the array row-wise yields faster memory access to a sequence of cells compared to alternative access patterns. The shortest path algorithm extension to the flood-fill algorithm is cache-efficient. In contrast to Dijkstras' algorithm or A* with consistent estimates, however, it can lead to many *reopenings* as cells may be expanded before their shortest path distance value has settled. In other words, the flood-fill algorithm is not optimal in the number of node expansions.

We know that, in the worst case, the number of re-openings in a graph in A* with inconsistent heuristic estimates can be exponential in the size of the state space and thus lead to an exponential number of node visits! Hence, one question is how worse the running time of Flood-Fill algorithm (for computing shortest paths) can be.

Given a gridworld with v cells. There are examples for which the flood-fill algorithm requires $\Omega(v^2)$ time in the worst case. Take the following grid with $3k$ cells $n_{i,j}$ for $1 \leq i \leq 3$ and $1 \leq j \leq k$

L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	…	L
@		@		@		@		@		@		@		@		@		@		@		@		@		@	…	@
S																											…	

Cells $n_{1,j}$, $j \geq 1$, in the first row are lava and incur a cost of 2, cells in the second row alternate between obstacles ($n_{2,2j-1}$) and being free ($n_{2,2j}$) with a cost of 1. Cells $n_{3,j}$ in the third row are all free. Start is cell $n_{3,1}$. Now the Flood-Fill algorithm traverses the third row completely, inserting all the freespace cells in between the obstacles into the queue. Processing these entries from left to right leads to a continuous improvement, as the new path to a lava-labeled cell in the first row will always be shorter than an existing one. Hence, the first row is scanned a linear number of times, for a quadratic number of expansions in total.

Given that edge costs are constant and that for each pass to the queue at least one node on the optimal path becomes settled (as it is in the Bellman-Ford algorithm), we have strong arguments that the complexity remains polynomial. The lack of a linear worst-case complexity bound, however, calls for alternatives.

Moreover, the flood-fill algorithm is less flexible than a graph-based solution which is applicable to several other workspace decompositions (triangulations, trapezoidal maps) in robot motion planning.

However, traditional graph search implementations are often too slow. For graph search in the PTSP, the lack of performance is the main reason to apply either flood-filling algorithms, or to rely on grid approximations. To finish the preprocessing within startup time, the default implementation approximates the input graph, in which A* searches pairwise shortest paths. One additional drawback about approximation is that for every non-cached entry, grid cells must be mapped to the graph, leading to a nearest neighbor search.

There is a refined implementation of the SSSP algorithm of Dijkstra (and A* with a consistent heuristic) for the PTSP that has optimal worst-case linear-time complexity. We use an implementation for Dijkstra's SSSP algorithm based on radix heaps. For v nodes and e edges in the graph, the running time for completely annotating a graph with minimal start-to-node distances is $O(e + v \log C)$, where C is the maximum weight (a.k.a. cost, length) of an edge. In other words, weights are values in $\{1, \ldots, C\}$.

To stay within integer range (the radix heap also works for floating point numbers, but $\log C$ is larger) for the octile gridworld of the PTSP, we use the approximation $577/408$ of $\sqrt{2}$. Then we multiply the cost of each node by 408 and divide the solution cost by the same value. Even for thousands of search nodes, the difference $|\sqrt{2} - 577/408| = 0.0000021239$ is small enough not to influence the outcome. This keeps C sufficiently low, so that $\log C$ can be considered as a constant (64 on a 64-bit computer), for an optimal $\Theta(e+v) = \Theta(v)$ worst-case time performance (the number of edges is at most eight times the number of nodes). For the PTSP we have the weights are positive integer numbers small enough not to exceed the integer representation for accumulated distances at each node.

23.5 TSP Search

The name PTSP indicates that the problem of serving an initial list of waypoints to the controller is a TSP. TSP is NP-hard in the *strong sense*, which makes it hard to approximate. For a polynomial q let π_q be the restriction of π to inputs I s.t. $\max(I) \leq q(|I|)$, where $|I|$ is the length of I. Then π is strongly NP-hard if π_q is NP-hard. The problem π_q for the TSP is the NP-hard *Hamiltonian path problem*, where a graph on n nodes has a Hamiltonian path, if its longest path has length $n-1$. This means for a growing number of waypoints n it is unlikely that a polynomial-time algorithm exists. Inapproximability results, however, must be taken with care: if the triangular inequality holds (such as in the Euclidean plane): using Christofides' algorithm the TSP can be approximated with factor 1.5.

For TSP good lower bounds are known. For the generalized TSP, GTSP for short, tight lower bounds are more difficult to derive. As the GTSP generalizes the TSP (immediate by setting $k = n$) it inherits its strong NP-hardness. Let us look at the number of possible simple paths in a graph that form possible tours. For $k = n$ colors this is the original TSP and we count $n!$ possible tours. This does not mean that every algorithm for solving the TSP has to have a complexity of $n! \approx \sqrt{(2\pi n)}(n/e)^n$. There is a dynamic algorithm with complexity $O(n^2 \cdot 2^n)$ as follows. Define $S \subseteq M$, $M = \{2,\ldots,n\}$ and $F(S,i)$ to be the shortest route starting at 1, ending at i that visits each waypoint in S exactly once. Then, for $|S| = \{i\}$, we have $F(\{i\},i) = d_{1,i}$ and $F(S,i) = \min_{j \in S, j \neq i}\{F(S \setminus \{i\}, j) + d_{j,i}\}$ with final answer $\min_{2 \leq i \leq n}\{F(M,i) + d_{i,1}\}$.

Similarly to the TSP, the number of tours does not neccessarily determine the runtime. We briefly describe a dynamic programming approach to find a colored tour (a simple path of mutually disjoint colors) for the GTSP. We are only interested in the decision problem, to find out whether or not a colorful simple path exists. We fill a table $T(s,t,C)$ (with values of type Boolean), where s and t are the source and the target locations and C is the set of colors of size i. We set $T(s,t,C) = \mathit{true}$ if and only if there is a tour of length i from s to t. For $i = 1$ we have $C = \{c\}$ so that $T(s,t,C) = \mathit{true}$ if and only if $t = s$ with color c. For $i = 2$ we have $C = \{c_1, c_2\}$ so that $T(s,t,C) = \mathit{true}$ if and only if there is an edge (s,t) in the graph with color c_1 for s and color c_2 for t. For $i+1$ colors, we traverse the ith level. For each cell w marked *true*, we look at the successors of w. If its color c does not appear in C, we mark $T(s,t,C \cup \{c\}) = \mathit{true}$. If there is a cell marked true in the last row, a colored tour has been found.

Comparing a TSP on $(n/2)$ waypoints with a GTSP of $(n/2)$ colors, shows that there are a factor 2^k more tours in the first problem than in the second. However, in terms of possible tours, a TSP on n colors is larger than a GTSP with $(n/2)$ colors, given that $n!/(n/2)! = n \cdot (n-1) \cdot \ldots \cdot (n/2+1) \geq 2^{n/2}$ for $n \geq 2$. For $k = n/c$ colors there are a factor $(n/k)^k$ more tours in the GTSP than in a TSP of k waypoints. In summary, the smaller the k, the smaller the number of tours, but if we fix the number of waypoints in the TSP to the number of colors allowed in the GTSP, then the second problem is harder.

23.5.1 Optimal TSP Solving

As the default TSP solver was too slow for solving PTSPs with $n \geq 30$ waypoints, we implemented an incremental depth-first branch-and-bound (DFBnB) planner.

As one heuristic we use the relaxation of the TSP. The Hungarian algorithm (HA) solves the *assignment problem* (AP, see Programs 23.1 and 23.2). It runs in cubic time if called from scratch and in quadratic time if computed incrementally. For a small number of waypoints $n \approx 15$ blind search is the fastest way to compute the best tour. For $n \approx 50$, HA is good enough to cut off branches in the DFBnB tree to help the solver to come up with an optimal tour. Compromises are weaker bounds that take less time. For the GTSP weaker heuristics apply.

Let $\lceil n/w \rceil = O(1)$. The DFBnB (G)TSP solver is optimal and, besides the efforts for computing the heuristic, has a constant-time worst-case performance at each generated search node. Space is at most quadratic in n.

The DFBnB solver avoids recursion and is based on a pre-allocated stack. This way all space needed is allocated prior to the search. With maximum depth n and at most n successors at a search node the memory requirements are $O(n^2)$. The trick to avoid duplicating the vector of waypoints already visited on a backtrack is to employ a bitvector. In the $w = 64$-bit computer era for current benchmark sizes it is safe to assume $\lceil n/w \rceil = O(1)$.

To establish $O(1)$ incremental time for the work at each search node in the GTSP we pre-compute an array of bitvectors *masks* $m[i]$, one for each color $i \in \{1,\ldots,k\}$, with $m[i]_j = 1$ iff waypoint G_j, $j \in \{1,\ldots,n\}$, has color i. If color i is visited during a depth-first traversal, then we use the update vector seen using the or bitvector operation, i.e., we set $seen = seen \vee m_i$ to change the status of all waypoints of color i. As bitvector operations are assumed to be constant time operations, the update time at each search node remains constant.

If more and more constraints are added, the lower bound from the AP is no longer tight. Examples from vehicle routing are time windows or capacity constraints. In the PTSP, we have that the pairwise distance table between two waypoints is not sufficient to predict the system behavior and that objectives like the change in the angle are becoming more prominent factors. Tracking angular change involves a computation for three waypoints and affects the accuracy of the lower bound, so that finding the optimal plan for the physically enhanced cost function becomes much harder.

Program 23.1: Hungarian method to solve assignment problem (first part).

```
public int HungarianMethod(int g, int city, int depth) {
int c = N - depth, ci = 0, count = 0, minval = 0, zero0 = 0, zero1 = 0;
      for (int i=0; i<c; i++)
        for (int j=0; j<c; j++)
            mask[i][j] = cost[i][j] = copy[i][j] = 0;
      for (int i=0; i<c; i++) rowCover[i] = colCover[i] = 0;
      for (int i=0; i<N; i++) {
        if (((used >> i) & 1L) > 0) continue;
        int cj = 0;
        for (int j=0; j<N; j++) {
              if (((used >> j) & 1L) > 0) continue;
              copy[ci][cj] = cost[ci][cj] = dist[i][j];
              if (i == j) copy[ci][cj] = cost[ci][cj] = Integer.MAX_VALUE;
              if (start != city && i == start) {
                if (j == city) copy[ci][cj] = cost[ci][cj] = g;
                else copy[ci][cj] = cost[ci][cj] = Integer.MAX_VALUE;
              }
              cj++;
        }
        ci++;
      }
      int step = 1; boolean finished = false;
      while (!finished) {
        switch (step) {
        case 1:
              for (int i=0; i<c; i++) {
                minval=cost[i][0];
                for (int j=0; j<c; j++) if (minval>cost[i][j]) minval = cost[i][j];
                for (int j=0; j<c; j++) cost[i][j] -= minval;
              }
              for (int i=0; i<c; i++)
                for (int j=0; j<c; j++)
                      if ((cost[i][j]==0) && (colCover[j]==0) && (rowCover[i]==0)) {
                        mask[i][j] = colCover[j] = rowCover[i] = 1;
                      }
              for (int i=0; i<c; i++) rowCover[i] = colCover[i] = 0;
              step = 3; break;
        case 3:
              for (int i=0; i<c; i++)
                  for (int j=0; j<c; j++)
                      if (mask[i][j] == 1) colCover[j]=1;
              count=0;
              for (int j=0; j<c; j++) count += colCover[j];
              if (count >= c) finished = true; else step = 4;
              break;
        case 4:
              int[] row_col = new int[2];
              while (true) {
                row_col[0] = -1; row_col[1] = 0;
                int i = 0; boolean done = false;
                while (!done) {
                    int j = 0;
                    while (j < c) {
                      if (cost[i][j]==0 && rowCover[i]==0 && colCover[j]==0) {
                            row_col[0] = i; row_col[1] = j; done = true;
                      }
                      j++;
                    }
                    i++;
                    if (i >= c) done = true;
                }
                if (row_col[0] == -1) { step = 6; break; }
```

Program 23.2: Hungarian method to solve assignment problem (second part).

```
                else {
                    mask[row_col[0]][row_col[1]] = 2;
                    boolean starInRow = false;
                    for (int j=0; j<c; j++)
                      if (mask[row_col[0]][j]==1) { starInRow = true; row_col[1] = j; }
                    if (starInRow==true) { rowCover[row_col[0]] = 1; colCover[row_col[1]] =
                        0; }
                    else { zero0 = row_col[0]; zero1 = row_col[1]; step = 5; break; }
                }
              }
              break;
          case 5:
              count = 0;
              int[][] path = new int[c*c][2];
              path[count][0] = zero0; path[count][1] = zero1;
              boolean done = false;
              while (!done) {
                int r = -1;
                for (int i=0; i<c; i++) if (mask[i][path[count][1]]==1) r = i;
                if (r>=0) {
                    count++;
                    path[count][0] = r; path[count][1] = path[count-1][1];
                }
                else done = true;
                if (!done) {
                    int t = -1;
                    for (int j=0; j<c; j++) if (mask[path[count][0]][j]==2) t = j;
                    count++;
                    path[count][0] = path[count-1][0]; path[count][1] = t;
                }
              }
              for (int i=0; i<=count; i++) {
                if (mask[(path[i][0])][(path[i][1])]==1) mask[(path[i][0])][(path[i][1])] = 0;
                else mask[(path[i][0])][(path[i][1])] = 1;
              }
              for (int i=0; i<c; i++) rowCover[i] = colCover[i] = 0;
              for (int i=0; i<c; i++)
                for (int j=0; j<c; j++)
                    if (mask[i][j]==2) mask[i][j] = 0;
              step = 3; break;
          case 6:
              minval = Integer.MAX_VALUE;
              for (int i=0; i<c; i++)
                for (int j=0; j<c; j++)
                  if (rowCover[i]==0 && colCover[j]==0 && (minval > cost[i][j]))
                    minval = cost[i][j];
              for (int i=0; i<c; i++) {
                for (int j=0; j<c; j++) {
                    if (rowCover[i]==1) cost[i][j] += minval;
                    if (colCover[j]==0) cost[i][j] -= minval;
                }
              }
              step = 4; break;
          }
        }
        for (int i=0; i<c; i++)
          for (int j=0; j<c; j++)
              if (mask[i][j] == 1) { assignment[i][0] = i; assignment[i][1] = j; }
        int sum = 0;
        for (int i=0; i<c; i++)
          sum += copy[assignment[i][0]][assignment[i][1]];
        return sum - g;
    }
}
```

23.5.2 Suboptimal TSP Solving

For effective suboptimal TSP solving, we distinguish between two major search options: local search and Monte-Carlo search, the latter being a class of randomized tree search algorithms that back up values from the leaves of the search tree back to the decision nodes to direct the search towards the best plan found, while maintaining exploration breadth.

Local Search relies on tour mutation operators (like 3-OPT) or refined node insertion schemes. The alternative is *nested rollout policy adaptation* (NRPA), an extension to *nested Monte-Carlo search* (NMC). In Chapter 5 we have learnt NMC to be a recursive algorithm that performs a certain number of rollouts, where a rollout is a random path in the search tree starting from the root and ending at a leaf that can be evaluated to some score value. The search method in NMC takes the level l as an argument and decrements the value by 1 in every recursive call. If the value has decreased to 1 (or to 0 depending on the implementation), a rollout is initiated. At each choice point of a rollout the algorithm chooses the successor that gives the best score when followed by a single random rollout. Similarly, for a rollout of level l it chooses the successor node that gives the best score when followed by a rollout of level $l-1$. Roughly speaking, the search intensifies with increasing recursion depth.

Policy adaptation leads to considerable performance improvements. Rather than navigating the tree directly, NRPA instead uses gradient ascent on the (Boltzman softmax) rollout policy at each level of the search. The planner learns a policy in form of a likelihood mapping from going from one waypoint to another. The policy is initialized to zero and is adapted each time a tour improvement has been found. Moreover, by the virtue of the nestedness of the search, the policy tables are refreshed (one table acts in each level of the search), so that we obtain a compromise between exploration and exploitation during the search. We observed that NRPA is more efficient than the backtrack solver for TSP problems with $n > 40$.

The NRPA algorithm selects, evaluates, and backs up random tours, adapting policies in form of $n \times n$ matrices (stored globally) at each of the at most l levels of the search. Besides backing up a tour at the end of each rollout, NRPA allocates no additional space during the search, so that in each level of the search only one best tour is active. Thus, for a level l search and n waypoints, the (G)TSP NRPA solver requires $O(ln^2)$ space.

It is, however, necessary to enhance the TSP solver with processing information on the inherent physical constraints in the PTSP. It is not difficult to observe that the sign of the determinant

$$D(p,q,r) = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

determines whether $r = (r_x, r_y)$ lies left or right of the line defined by $p = (p_x, p_y)$ and $q = (q_x, q_y)$. Moreover, $|D(p,q,r)|$ in fact is twice the surface of the triangle $T = (p,q,r)$ determined by p, q, and r. We use $|D(p,q,r)|$ as one term to our cost function to support the observation that small surfaces lead to good steering behavior of the ship. The surface spanned for a path of projected states $p = (c_1 = \phi(s_1), \ldots, c_k = \phi(s_k))$ is $\sum_{i=2}^{k} |D(c_{i-2}, c_{i-1}, c_i)|$. By applying Sarrus' rule we have $|D(p,q,r)| = |q_x r_y + p_x q_y + p_y r_x - p_y q_x - p_x r_y - q_y r_x|$, which shows that the computation itself is simple.

The triangle surface supports the two concepts applied, *angular change* and *indirectness*, where the latter refers to the difference in path length between two waypoints of the computed path distance to the straight-line distance in the Euclidean metric space.

A compromise between the accumulated path distance value δ and path surface value Δ had to be found. For surpassing *lava* we multiplied the edge weight to a grid cell by some constant α: the higher α the more the ship avoids crossing the lava area.

Once the order of waypoints to be visited has been computed, it is forwarded to the controller, which then aims at following it. The controller needs some limited lookahead to allow smooth steering. In the default

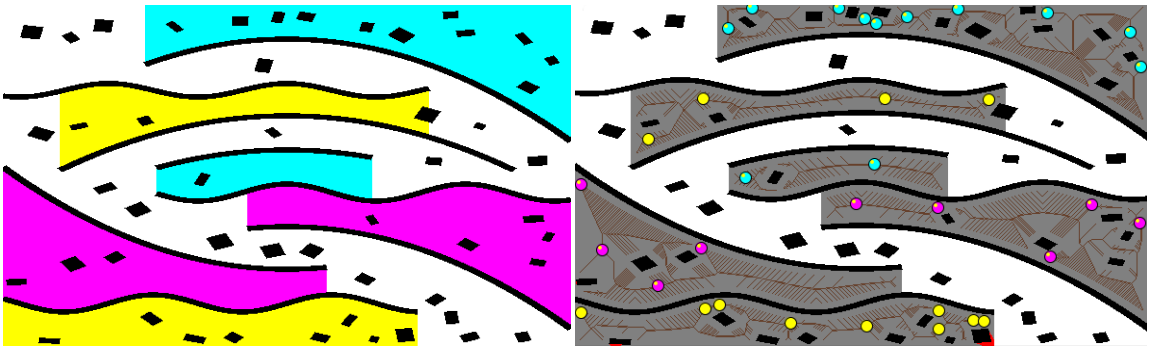


Figure 23.4: Input for the colored inspection problem (left). Approximation of the medial axis generated by the grass-firing transformation algorithm as well as the inspection points generated by the approach (right). The red areas are not visible from the inspection points (inspection quality was set to $\alpha = 0.99$).

implementation this is done by issuing random searches with respect to multistep operations (called *macro actions*). Within a macro action, no change in the already selected action proposal is allowed.

In an existing policy P rollout children s' for a node s are chosen with respect to $e^{P(s,s')}$. The choice of successors is done using a roulette wheel fitness selection based on these values. Initially, all policy values are set to zero. As the entire state-to-state table surely is too big, it is projected to an essential part to be learnt. In the standard TSP this is a mapping from waypoint to waypoint, while for the controller we have a mapping from the path position to the macro action to be chosen. Given a tour that improves the current best cost value, policy adaptation now performs gradient descent to ensure that the probability of a good tour increases.

Hence, we included policy adaptation in the controller. Due to the limited time of execution, we decided not to rely on the outcome of an NMC. Each time a new improving random path with respect to the existing policy (initial random) is computed, a (Bellman) update is issued to update the policy. Since computation time is a scarce resource, when executing a macro action (where no decision takes place) the remaining computing time is used to improve the policy. This option already present in the pure random controller reminds of a chess playing program that analyzes different lines of play also at the opponent's turn.

23.6 Inspection Problem

For solving the inspection problem, we have to find suitable waypoints automatically.

In our first setting we assume the environment to be specified as a bitmap image $\mathscr{I}$ consisting of obstacles $O_1, \ldots, O_m$ and regions $R = \{R_1, \ldots, R_n\}$ that should be inspected, as shown in Figure 23.4. Each region $R_i \in R$ defines a contiguous area in $\mathscr{I}$ which does not intersect any other region or obstacle. Obstacles are colored in ■ and the unoccupied area $U = \mathscr{I} \setminus (R_1 \cup \ldots \cup R_n \cup O_1 \cup \ldots \cup O_m)$ is colored in □. The user specifies the color of each region $R_i \in R$, denoted by $\text{color}(R_i)$. The only requirement is that $\text{color}(R_i)$ is neither ■ nor □. This provides a general definition as it allows for the same color to be used for different regions. The set of all region colors is defined as $\text{colors}(R) = \{\text{color}(R_i) : R_i \in R\}$. For a point $p \in \mathscr{I}$, $\text{color}(p)$ denotes its color.

Colors allow the user to group regions and specify desired constraints on how the robot should carry out the inspection. In particular, the robot is required to inspect all the regions in one color group before inspecting the region from another, where a color $c \in \text{colors}(R)$ defines the color group $\{R_i \in R : \text{color}(R_i) = c\}$.

The robot carries out the inspection by taking snapshots at different locations, and the inspection points should be automatically computed. The robot can take a snapshot of a region $R_i \in R$ only when it is inside R_i. As a result, a point $q \in R_i$ is considered visible from $p \in \mathscr{I}$ iff $\overline{pq} \in R_i$, i.e., the straight-line segment from p to q remains entirely in R_i. Let $\text{vis}(R_i, p)$ denote the area of R_i that is visible from a point $p \in \mathscr{I}$, i.e., $\text{vis}(R_i, p) = \{q \in R_i : \overline{pq} \in R_i\}$. Note that $\text{vis}(R_i, p) = \emptyset$ when $p \notin R_i$. The area of R_i visible from a collection of points $p_1, \ldots, p_\ell \in \mathscr{I}$

is defined as $\text{vis}(R_i, p_1, \ldots, p_\ell) = \bigcup_{j=1}^{\ell} \text{vis}(R_i, p_j)$. When considering all the regions in R, the visible area is defined as $\text{vis}(R, p_1, \ldots, p_\ell) = \bigcup_{R_i \in R} \text{vis}(R_i, p_1, \ldots, p_\ell)$. The quality of a collection of points $p_1, \ldots, p_\ell \in \mathscr{I}$ is defined in terms of the fraction of the area in R that it enables the robot to inspect, i.e.,

$$\text{quality}(R, p_1, \ldots, p_\ell) = \frac{\text{area}(\text{vis}(R, p_1, \ldots, p_\ell))}{\text{area}(\cup_{R_i \in R} R_i)}. \tag{23.1}$$

A tour, denoted by $\text{tour}(p_1, \ldots, p_\ell)$, defines an ordering of the points $p_1, \ldots, p_\ell$. The notation Γ_i, where $\Gamma = \text{tour}(p_1, \ldots, p_\ell)$, refers to the i-th point in the tour. The inspection problem can now be stated as follows: Given a desired inspection quality $0 < \alpha \leq 1$, a bitmap image $\mathscr{I}$ consisting of obstacles $O_1, \ldots, O_m$, regions $R = \{R_1, \ldots, R_n\}$, and the start point $p_* \in \mathscr{I}$, compute a collection of points $p_1, \ldots, p_\ell \in \mathscr{I}$ where the robot is going to take the additional snapshots to carry out the inspection task; a tour $\Gamma = \text{tour}(p_*, p_1, \ldots, p_\ell)$, which starts at p_* and specifies an ordering of the points $p_1, \ldots, p_\ell$; and a path, denoted by $\text{path}(\Gamma)$, which connects the points in the order defined by Γ;

such that $\text{quality}(R, p_*, p_1, \ldots, p_\ell) \geq \alpha$; $\text{path}(\Gamma)$ is collision free; and color groups are inspected in order, i.e., $\forall i, j, k \in \{1, \ldots, |\Gamma|\} : \square \notin \{\text{color}(\Gamma_i), \text{color}(\Gamma_j), \text{color}(\Gamma_k)\} \wedge i < j < k \wedge \text{color}(\Gamma_i) \neq \text{color}(\Gamma_j) \implies \text{color}(\Gamma_i) \neq \text{color}(\Gamma_k)$

The approach seeks to reduce the number of points $p_1, \ldots, p_\ell$ it needs to carry out the inspection task. To further reduce the distance traveled by the robot, the points in $\text{tour}(p_*, p_1, \ldots, p_\ell)$ will be connected with short collision-free paths.

23.7 Method

Pseudocode is shown in Algorithm 23.1. The approach starts by generating several inspection points on the medial axis of each region $R_i \in R$ to increase the visible area (Algorithm 23.1). The approach uses a grass-firing transformation algorithm to efficiently approximate the medial axis.

A key aspect of the approach is the transformation of the inspection problem into CTSP over an undirected, colored, and weighted graph $G = (V, E, \text{color}, \text{cost})$. More specifically, the vertex set V consists of the inspection points generated by the approach, and the start point p_* (Algorithm 23.1). Each $p \in V$ inherits the color of the region that contains it, i.e., if $p \in R_i$ then $\text{color}(p) = \text{color}(R_i)$; if $p \in U$ then $\text{color}(p) = \square$.

The set of edges contains all pairs, i.e., $E = V \times V$. The cost of an edge $(p_1, p_2) \in E$ is defined as the length of the shortest path from p_1 to p_2. An optimized implementation of Dijkstra's shortest-path algorithm is used to compute the all-pairs shortest paths (Algorithm 23.1).

A branch-and-bound search is developed to solve the resulting CTSP (Algorithm 23.1) by computing a low-cost tour Γ which starts at p_*, visits each vertex in V, and inspects the clustered groups in order. After computing Γ, the approach relies on a controller to move the robot from one inspection point to the next as specified by Γ (Algorithm 23.1).

The rest of the section describes the main steps of the approach in more detail.

23.7.1 Generating the Inspection Points

Pseudocode for generating the inspection points is shown in Algorithm 23.2. In order to increase the visible area, the approach seeks to generate the inspection points on the medial axis of each region $R_i \in R$. In a discrete

Algorithm 23.1: Pseudocode for inspection

Input: α: inspection quality, $0 < \alpha \leq 1$; bitmap image $\mathscr{I}$ consisting of obstacles $O = \{O_1, \ldots, O_m\}$, regions $R = \{R_1, \ldots, R_n\}$ that should be inspected; $p_* \in \mathscr{I}$: start point
Output: a short, collision-free, path that solves the inspection problem

inspectionPts $\leftarrow$ *GenerateInspectionPoints*$(\mathscr{I}, \alpha)$
$G = (V, E, \text{color}, \text{cost})$, where $V \leftarrow \text{inspectionPts} \cup \{p_*\}$ and $E \leftarrow V \times V$
data $\leftarrow$ *AllPairsShortestPaths*$(\mathscr{I}, V)$
for $(p_1, p_2) \in E$ **do**
  $\text{cost}(p_1, p_2) \leftarrow$ *length*(*RetrievePath*(data, p_1, p_2))
$\langle p_1, \ldots, p_r \rangle \leftarrow$ *CTSPsolver*(G, p_*)
FollowTourVehicleController$(p_1, \ldots, p_r)$

Algorithm 23.2: *GenerateInspectionPts*$(\mathscr{I}, \alpha)$

Input: $\mathscr{I}$: bitmap image; α: desired inspection quality, $0 < \alpha \leq 1$
Output: a set of inspection points

$h \leftarrow \text{height}(\mathscr{I})$; $w \leftarrow \text{width}(\mathscr{I})$
$\mathscr{B} \leftarrow \text{zeros}(h, w)$ {grass-fire transformation}
for $(i, j) \in \{0, \ldots, h-1\} \times \{0, \ldots, w-1\}$ **do**
  if $\text{color}(\mathscr{I}(i, j)) \notin \{\square, \blacksquare\}$ **then**
    $\mathscr{B}(i, j) \leftarrow 1 + \min\{\mathscr{B}(i-1, j), \mathscr{B}(i, j-1)\}$
for $(i, j) \in \{h-1, \ldots, 0\} \times \{w-1, \ldots, 0\}$ **do**
  if $\text{color}(\mathscr{I}(i, j)) \notin \{\square, \blacksquare\}$ **then**
    $\mathscr{B}(i, j) \leftarrow 1 + \min\{\mathscr{B}(i+1, j), \mathscr{B}(i, j+1)\}$
skeleton $\leftarrow$ extract most intense pixels in the brightness map $\mathscr{B}$ {select inspection points}
skeleton $\leftarrow$ *filter*(skeleton)
inspectionPts $\leftarrow$ skeleton
currScore $\leftarrow$ *VisScore*$(\mathscr{I}, \text{inspectionPts})$
for $p \in$ skeleton **do**
  newScore $\leftarrow$ *VisScore*$(\mathscr{I}, \text{inspectionPts} \setminus \{p\})$
  if newScore $\geq \alpha \vee$ currScore $=$ newScore **then**
    inspectionPts $\leftarrow$ inspectionPts $\setminus \{p\}$
    currScore $\leftarrow$ newScore
return inspectionPts

grid, e.g., as imposed by the bitmap image $\mathscr{I}$, it is possible to approximate the medial axis by applying skeletal algorithms. One particularly efficient approach is the grass-firing transformation algorithm which uses two passes over $\mathscr{I}$ to compute the Manhattan distances from the obstacles for each pixel (Algorithm 23.2). Interpreting distances as brightness, the medial axis approximation is obtained by extracting the skeleton consisting of the locally brightest pixels (Algorithm 23.2). Figure 23.4 shows an example of the medial axis computed by the grass-firing transformation.

The points on the medial-axis skeleton are filtered to determine viable candidates for the inspection points (Algorithm 23.2). The filtering process removes points that are too close to the obstacles, imposes some minimum separation among inspection points, and gives preference to branching points, which have a degree of 3 or more in the skeleton graph. In our examples, the filtering process brought down the number of inspection points from thousands to 100–200. This is still a large number that would impose considerable runtime requirements on the CTSP solver and the vehicle controller.

To further reduce the number of inspection points while maintaining the desired inspection quality, candidates are also filtered according to the overall visibility score. The visibility score for a collection of points provides an efficient approximation of the inspection quality measure (Eqn. 23.1). The process we apply computes the visible area for each $p \in$ inspectionPts by casting light rays in all eight grid directions. To account for the color constraints, the light intensity becomes zero when encountering an obstacle or leaving the region $R_i \in R$ that

contains p. The lightmaps computed for each $p \in$ inspectionPts are merged into an overall visibility map, where each $q \in \mathscr{I}$ is marked as visible when it is visible from at least one inspection point. $VisScore(\mathscr{I}, \text{inspectionPts})$ (Algorithm 23.2) is then computed by dividing the number of visible pixels by the number of pixels corresponding to regions in R.

The visibility score is used to further reduce the number of inspection points while maintaining the inspection quality (Algorithm 23.2). For each candidate inspection point $p \in$ skeleton, we compare the visibility score with and without p. If the visibility score for inspectionPts $\setminus \{p\}$ is at least α, then p is removed from the inspection points since the inspection quality is still high. The point p is also removed when it does not impact the visibility score, i.e., it remains the same for inspectionPts $\setminus \{p\}$ and inspectionPts.

23.7.2 CTSP Solver

As described earlier, the inspection problem is transformed into a CTSP over an undirected, colored, and weighted graph $G = (V, E, \text{color}, \text{cost})$. Given that each vertex $p \in V$ inherits the color of the region that contains it, the objective is to compute a low-cost tour which starts at $p_* \in V$ and visits each vertex in V according to their color.

Formally, a colored tour is defined as follows. Let $G = (V, E, \text{color}, \text{cost})$ denote an undirected, colored, and weighted graph. Let $p_* \in V$ denote the start vertex. A sequence of vertices $\langle p_1, \ldots, p_r \rangle$ constitutes a valid colored tour if $\{p_1, \ldots, p_r\} = V$; $p_1 = p_*$; $\forall i \in \{1, \ldots, r-1\} : (p_i, p_{i+1}) \in E$; and $\forall i, j, k \in \{2, \ldots, r\} : i < j < k \wedge \text{color}(p_i) \neq \text{color}(p_j) \implies \text{color}(p_i) \neq \text{color}(p_k)$.

An *optimal clustered tour* is a colored tour with minimum cost, where the cost of the tour is defined as the sum of the weights associated with the edges of the tour.

Our CTSP solver is based on depth-first branch-and-bound search (*DFBnB*). For applying DFBnB to the problem, we extended depth-first search (DFS) with upper and lower bounds. In this context, branching corresponds to the generation of successors, so that DFBnB generates a *branch-and-bound* search tree. One way of obtaining a lower bound L for the problem state u is to apply an *admissible heuristic* h with $L(u) = g(u) + h(u)$, where g denotes the cost for reaching the current node from the root, and h is a function that always underestimates the remaining cost to reach a goal. As with standard DFS, the first solution obtained might not be optimal. With *depth-first branch-and-bound*, however, the solution quality improves over time together with the global value U until eventually the lower bound $L(u)$ at some node u is equal to U.

23.7.3 Following the CTSP Tour

Once a tour Γ is computed, a vehicle controller is employed to drive the robot from one inspection point to the next in the order defined by Γ. The vehicle is controlled by setting the acceleration (on or off) and turning the vehicle left, right, or keeping it straight. The state of the vehicle at time step t is described by the orientation d_t, velocity v_t, and position p_t. The equations of motions are defined as

$$d_{t+1} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} d_t, \tag{23.2}$$

$$v_{t+1} = (v_t + d_{t+1} a_t K) L, \tag{23.3}$$

$$p_{t+1} = p_t + v_{t+1}, \tag{23.4}$$

where θ is a fixed angle which determines the rotation at each time step (θ is 0 if the action is to go straight), K is the acceleration constant, a_t is 1 if the acceleration is on and 0 if it is off, and L is the friction loss factor.

Table 23.2: Solving (top) the standard inspection problem (bottom) the inspection problem with colored regions.

Map	Preprocess	IPs	Cost	Steps
01	0.95s ± 0.24s	4	767.5 ± 0	583.8 ± 11.5
02	0.45s ± 0.02s	6	1,171.5 ± 0	967.5 ± 84.9
08	0.32s ± 0.01s	4	1,965.4 ± 0	2,142.3 ± 76.3
19	1.69s ± 0.10s	13	2,451.6 ± 57.8	2,144.5 ± 76.0
24	0.91s ± 0.02s	12	1,818.4 ± 0	1,577.3 ± 27.7
35	0.57s ± 0.04s	11	1,478.4 ± 0	1,171 ± 50.5
40	0.38s ± 0.03s	5	975.8 ± 0	804 ± 16.0
45	0.68s ± 0.01s	12	1,792.1 ± 0	1,506.3 ± 24.5
61	0.72s ± 0.14s	1	123.3 ± 0	162 ± 6

Map	Preprocess	IPs	Cost	Steps
01	1.98s ± 0.01s	9	1,727.5 ± 0	1,387.7 ± 44.4
02	2.69s ± 0.12s	12	1,930.2 ± 0	1,543.8 ± 36.6
08	1.98s ± 0.05s	10	1,614.0 ± 0	1,422 ± 116.6
19	11.74s ± 0.45s	30	3,558.0 ± 55.5	4,324.1 ± 186.8
24	4.68s ± 0.24s	17	2,546.9 ± 0	2,187.4 ± 76.9
35	2.36s ± 0.08s	12	1,863.4 ± 10.3	1,607 ± 90.8
40	1.97s ± 0.02s	9	1,566.5 ± 0	1,380.6 ± 62.3
45	2.96s ± 0.03s	14	2,348.9 ± 7.9	2,323.2 ± 161.8
61	1.53s ± 0.08s	4	1,323.6 ± 0	1,082.3 ± 19.4

Columns indicate the following: (Map) map number; (Preprocess) time to process the input image, generate the inspection points, and compute the shortest paths (Algorithm 23.1); (IPs) number of inspection points generated by the approach; (Cost) estimated traveled distance when following the inspection tour; (Steps) the number of steps taken by the controller. Results are given averaged over 10 runs, ± standard deviation. Waypoint counts are the same for all runs of a map-problem pair, hence no standard deviation.

The vehicle inertia is preserved, which makes the navigation more challenging. The execution model of the controller is real-time. The controller has numerous applications ranging from modeling non-player characters in interactive games to approximating steering behavior of mobile robots. The testbed has raised significant interest in combined task and motion planning.

23.8 Evaluation

The PTSP system is implemented in Java to cooperate with the PTSP simulator.

23.8.1 PTSP Simulation

The approach is first evaluated in the PTSP using generated maps with random distribution of goal regions and obstacles, as shown in Figure 23.4. Experiments are conducted under two scenarios. In the first scenario, the robot is required to inspect the entire area not occupied by obstacles, i.e., $\mathscr{I} \setminus (O_1 \cup \ldots \cup O_m)$. This corresponds to the standard inspection problem where there is only one color. In the second scenario, the robot is required to inspect only the colored regions. Each region was assigned one of three colors. In all the experiments, the inspection quality α was set to 0.99. Every experiment is run 10 times; results report the average and standard deviation statistics.

Table 23.2 displays the results of the two sets of experiments: the standard inspection problem and the inspection problem with colored regions. The results show the approach works well under different scenarios and environments, generating low-cost tours. The runtime, which includes the time to process the input image, generate the waypoints, and compute the shortest paths (Algorithm 23.1), is about 1s for the standard inspection problems and generally between 1.5s and 3s for the inspection problems with colored regions.

The first competitor is *Random*. It constructs a random valid tour. For one version of DFBnB (denoted as DFBnB + AP) we imposed at most 10,000 node expansions and a heuristic based on the assignment problem (AP), while for the latter we applied at most 100 million node expansions and the trivial heuristic, always returning zero. The results of these comparisons, obtained as averages over three runs, are shown in the plots in Figures 23.5 to 23.7.

Note that the precomputation efforts (Algorithm 23.1) are the same for all the CTSP solvers. Results in Figure 23.5 show that *Random* is the fastest CTSP solver but, as shown in Figure 23.6, it generates high-cost

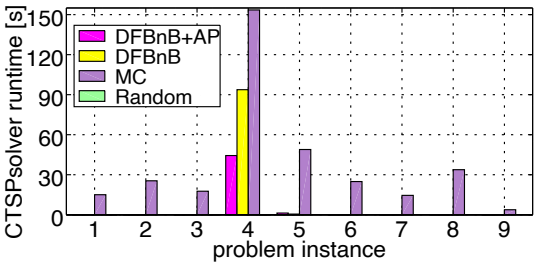


Figure 23.5: Mean runtime for each CTSPsolver to compute the CTSP tour. In many instances, the runtime was just a few milliseconds; hence, it is not visible in the graph.

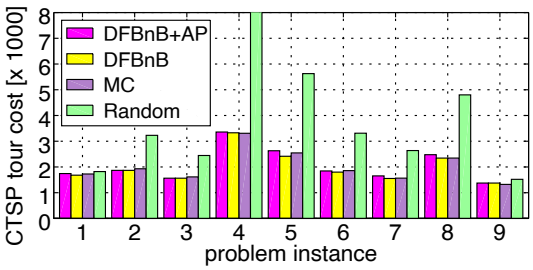


Figure 23.6: Mean tour cost as computed by each CTSP solver. Note slight difference in the cost function: MC processes floating point data, while DFBnB uses integers.

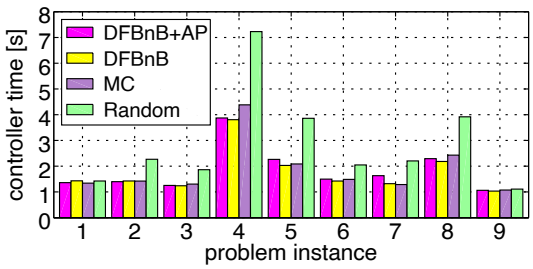


Figure 23.7: Mean number of steps taken by the controller when following the CTSP tour computed by each CTSP solver.

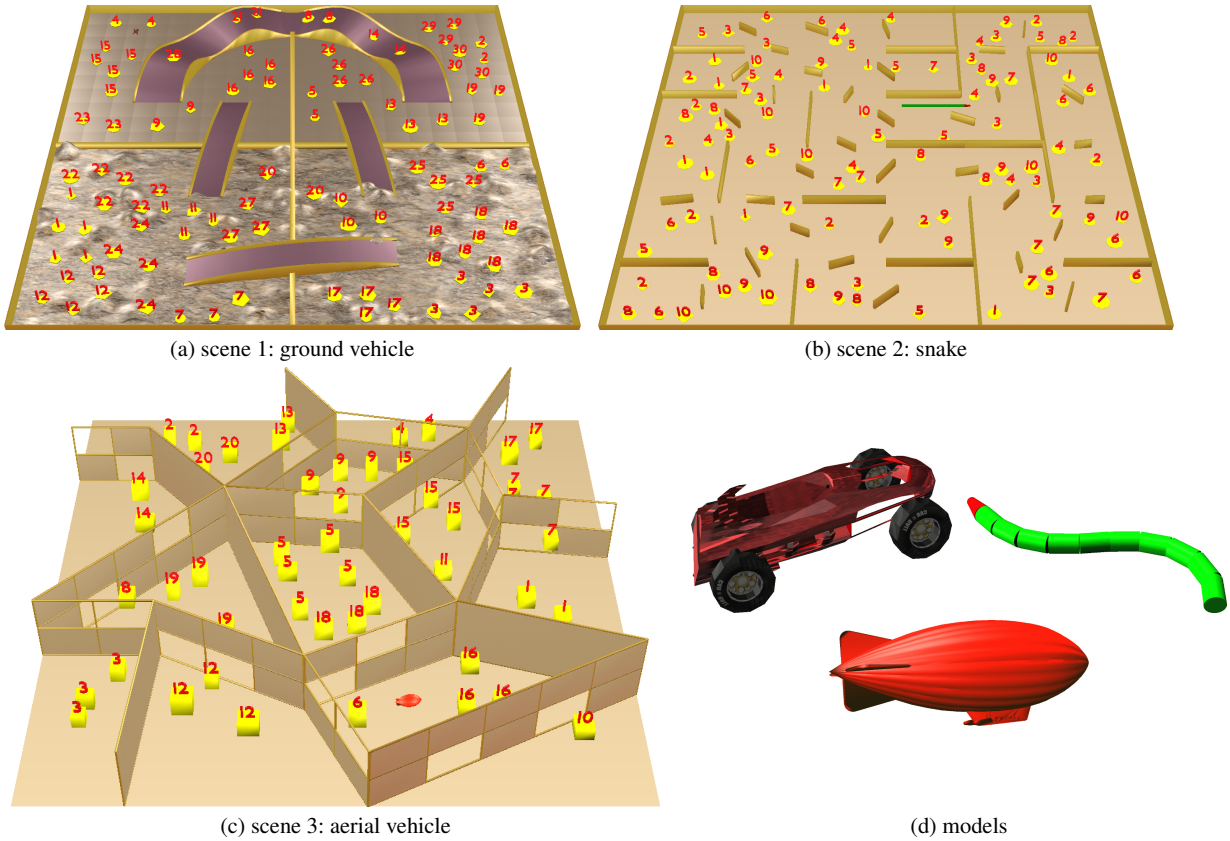
tours. In terms of costs (Figure 23.6), DFBnB is no clear-cut winner. This is also corroborated by the results in Figure 23.7 which shows that the number of steps taken by the simulator is similar when following the tours computed by DFBnB. DFBnB, however, can be much faster (see Figure 23.5), especially for smaller instances. DFBnB has a low variance as it is a deterministic algorithm.

When considering hard problem instances, the difference in the runtime for various algorithms can depend on the parameters chosen, namely the maximum number of expansions in DFBnB and the depth and width of the recursion tree in MC. These parameters provide a mechanism that can be tuned by the user in striking the right balance between runtime and solution cost.

23.8.2 Robot Simulation

As part of the problem is to find multiple goals in a complex environment, Figure 23.8 shows some problem domains and models we looked at in our robot simulation experiments.

Figure 23.9 shows the runtime results when varying the number of goals from five to 100 and the number of groups from 10 to 100, using clustering as well as random partitioning of goals into groups. The results show



(a) scene 1: ground vehicle

(b) scene 2: snake

(c) scene 3: aerial vehicle

(d) models

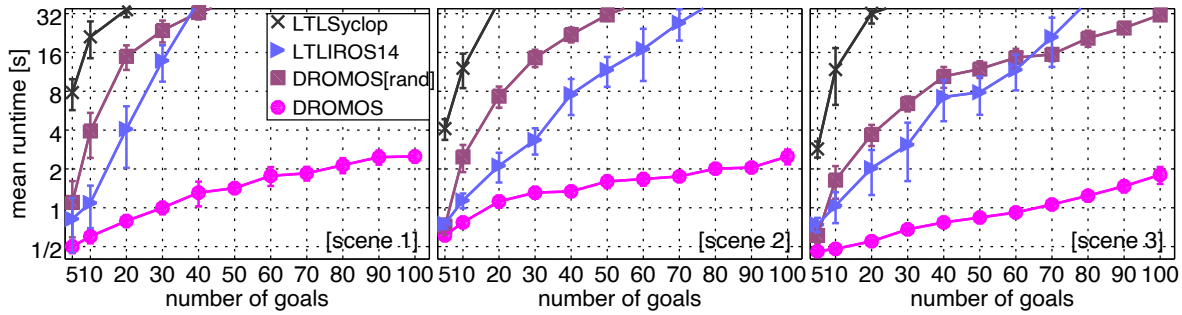
Figure 23.8: Scenes and models. Goal regions are labeled with the group id. The first scene shows 100 goals clustered into 30 groups. The second scene shows 100 goals partitioned at random into 10 groups of 10 goals per group. The third scene shows 50 goals partitioned into 20 groups. In the third scene, the aerial vehicle is not allowed to fly outside the boundaries or above the walls; the only way to go from one room to the next is to fly through the open windows.

that in all cases the introduced system (*Dromos*) is significantly faster than the other planners. Even in the case of 100 goals, *Dromos* can find a solution in 2 to 2.5 seconds. In contrast, an LTL motion planner suffers from scalability issues due to its reliance on the combination of the LTL automaton with the roadmap graph and the fixed order in which to visit the color groups as imposed by the LTL formula. In contrast, *Dromos* uses GTSP tours to guide the motion-tree expansion. When the expansion along a particular GTSP tour becomes too difficult, *Dromos* uses the selection penalties to promote expansion along alternative GTSP tours. Figure 23.9 also shows that *Dromos* is significantly faster than *Dromos*[rand], which uses random tours instead of *Dromos*. These results demonstrate the importance of using low-cost tours to guide the expansion of the motion tree.

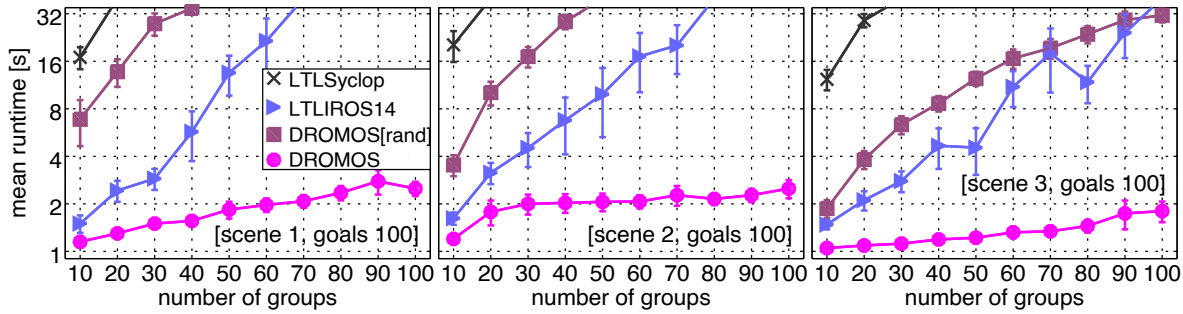
23.8.3 2D Inspection

Inspection experiments were carried out using several challenging 2D scenes, as shown in Figure 23.10, where the robot has to avoid numerous obstacles and pass through narrow passages to inspect the colored regions.

In a general inspection setting it is evident that even with a very large number of waypoints due to their size robots may not see everything (take a ball attached to a wall). Therefore, we transform the polygonal world into an image, and determine the medial axis in the resulting bitmap. While this can only result in an approximate result, the advantage of the approach is that it is robust even for complex obstacle shapes.



(a) TSP case where the number of groups is the same as the number of goals, i.e., one goal per group.



(b) GTSP case with 100 goals and the number of groups being varied as $10, 20, \ldots, 100$.

Figure 23.9: Results on the mean runtime. Due to significant differences in runtime, logscale is used for the y-axis with the label showing the actual value rather than its logarithm. Each bar indicates the standard deviation.

Experiments have been conducted and show the impact of the number of colors, inspection quality, and the importance of using CTSP tours to guide the motion-tree expansion.

23.8.4 3D Inspection

Inspection in 3D is involved, as visibility aspects become complex. One has to decide between volume and surface inspection (we take the latter), and between inside and outside inspection (we do both). In 3D the subject of interest for inspection is a mesh, while the 3D substitute for the 2D medial axis approximation is a skeleton (the medial axis may have facets in 3D).

There are range and angle limitations of the visibility sensors that are relevant in practice but, for the sake of simplicity, we assume an unlimited sensor range. We also restrict ourselves to generating waypoints of only one color. It is not difficult to extend the setting to include all these additional constraints. Moreover, we need a robot that can fly. A 3D quadcopter model for doing surface inspection is provided in Figure 23.11.

Figures 23.12 and 23.13 show inside and outside surface inspections of complex 3D models carried out by the quadcopter. The waypoints generated were passed as input to the sampling-based motion planner, which computed a collision-free and dynamically feasible motion trajectory that enabled the quadcopter to reach each waypoint. The running time for the motion planner was negligible.

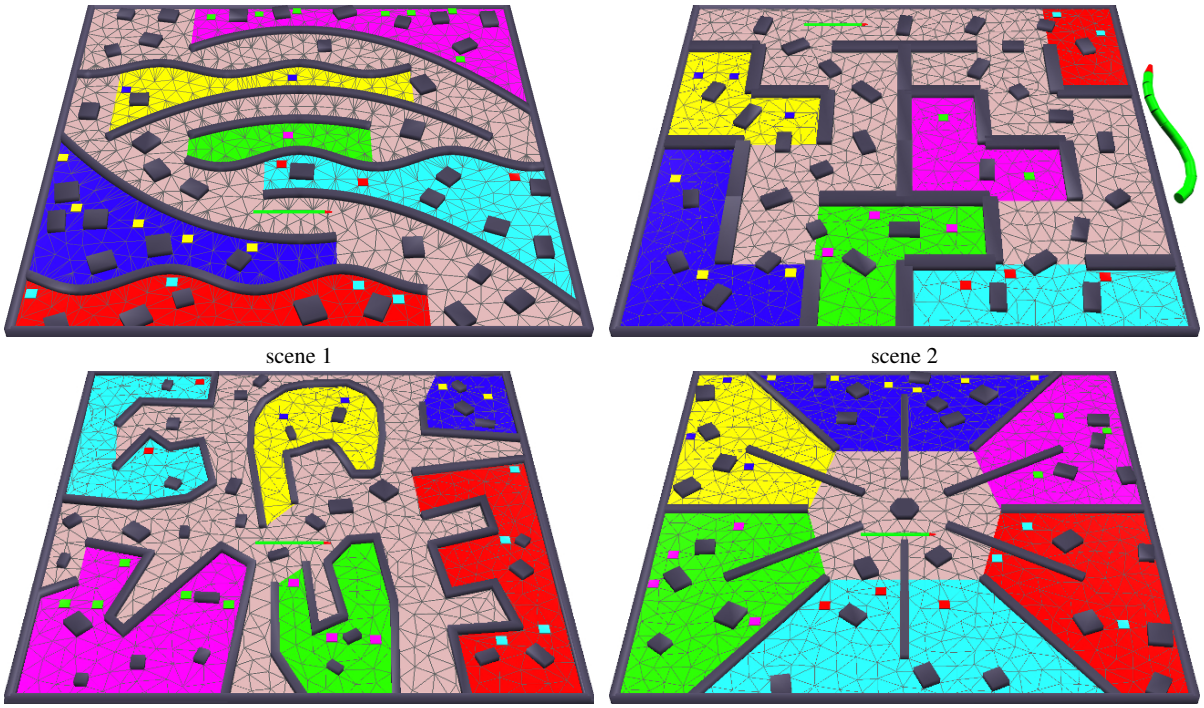


Figure 23.10: The snake-like robot model and scenes. Inspection points are shown as small squares colored with the inverse color of the region. The inspection points were automatically generated based on the desired inspection quality, set to 99.9%. The triangulation is also shown.

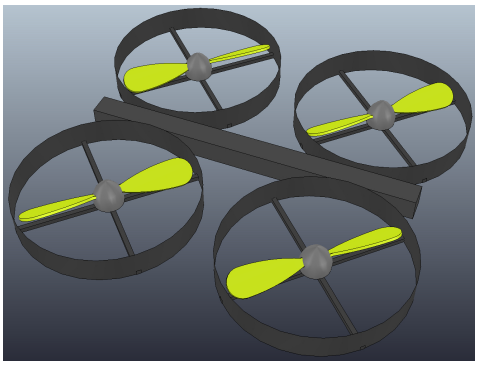


Figure 23.11: Physics-based model of the quadcopter used for the aerial inspection tasks.

23.9 Summary

The demands for a tight integration of task and motion planning in computer game playing and robotics have resulted in a rising research interest in the PTSP. We have presented and extended an existing PTSP engine. In both cases, we reduced the number of parameters of the search. We expect improved performance using machine learning to fine-tune the simulation parameters.

We have included efficient optimal and suboptimal TSP solvers, where the former is tuned for constant node performance and the latter exploits recent advances in Monte-Carlo tree search. For steering the ship in the simulator, we added policy adaptation. We also offered a new way of computing tours with a small change in direction through computing the surface of the ship's trajectories and used radix heaps for efficient SSSP search with the algorithm of Dijkstra.

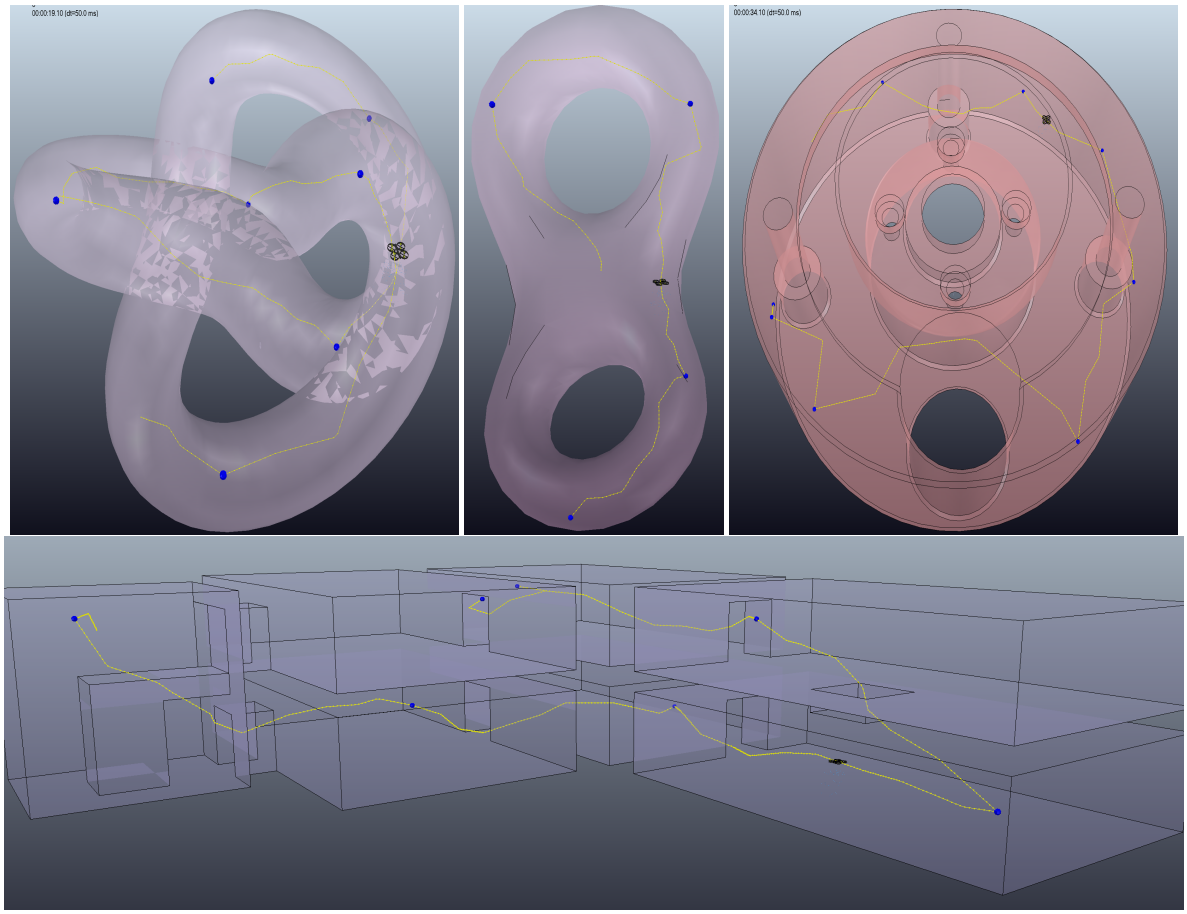


Figure 23.12: Scenes where the quadcopter is required to inspect the inside surface of the 3D model. The dots indicate the waypoints generated. The dotted line indicates the paths that the quadcopter needs to follow. After the waypoints were generated, the motion planner took 1/2 to 3 seconds to find a solution.

Using a graph for representing the grid helps to overcome the problem of the grid resolution being increased. Methods like Flood-Fill are bound to 2D, while the graph representation supported in the engineered shortest-path search is not.

We also developed an effective approach for the inspection problem with colored regions, integrated with sampling-based motion planning so that it can work for complex robotic systems with non-linear dynamics. The colored tours serve to guide the sampling-based motion planner, which in turn would provide information about the feasibility of each tour. A key aspect of the approach was the transformation of the inspection problem into a CTSP over an undirected, colored, and weighted graph. The approach relied on a grass-firing transformation algorithm to effectively approximate the medial axis of each colored region. The approach relied on ray casting and visibility scores to generate inspection points on the medial axis that would enable the robot to achieve the desired inspection quality. An effective CTSP solver was developed based on depth-first branch-and-bound search in to compute low-cost inspection tours. Experiments using different environments and inspection tasks provided promising validation.

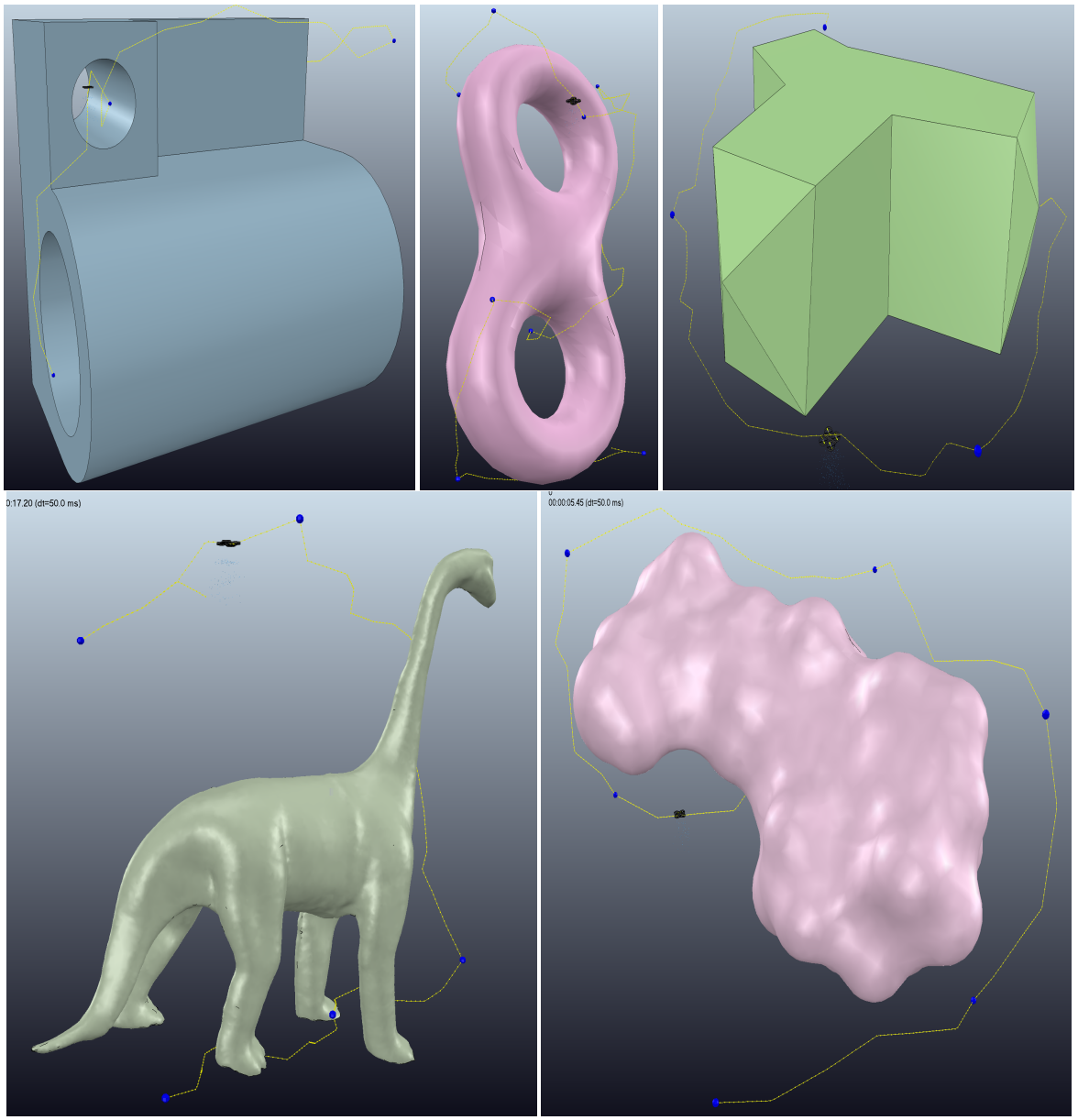


Figure 23.13: Scenes where the quadcopter is required to inspect the outside surface of the 3D model. The blue dots indicate the waypoints generated. The yellow dotted line indicates the paths that the quadcopter needs to follow. After the waypoints were generated, the motion planner took 0.5s-3s to find a solution.

23.10 Bibliographic Notes

Competitions at editions of the Conference on Computational Intelligence and Games (CIG) illustrated the progress in solving the physical traveling salesman problem. The winner of all PTSP competitions [509, 368] was the *Purofvio* system, developed by a research team from the University of York [508], with multi-objective PTSPs considered in [522]. The system applies Monte-Carlo search with macro actions and a TSP solver for choosing the waypoint ordering. Additionally, UCT [25] was adapted [510].

Depth-first branch-and-bound with applications to solve TSP is considered in [670, 671]. Even though the discrete setting simplifies significantly from high-dimensional non-linear motion planning for robotics con-

trol [515], successful solutions can serve as heuristics and to improve computer game play, autonomous logistics, and underwater inspection [242].

For a detailed presentation of the scan-line flood-fill algorithm along with the pseudo-code we refer the reader to [508]. By utilizing a queue, Flood-Fill shares similarity with the Bellman-Ford algorithm for computing shortest paths via iterated scans over the search graphs [136]. There are *speed-up techniques* for shortest path search [323, 412, 638, 36].

The Hungarian algorithm to solve the assignment problem is studied by [371]. From SAT solvers, however, we know that NP-hardness results for the worst-case time complexity alone is not necessarily the end of the success story [631]. This is also true to some extent for unconstrained TSPs. By using tight lower bounds close-to-optimal solutions for thousands of waypoints have been found [371].

The first transformation from GTSP into the TSP was introduced by Lien et al. [451]. Later, Dimitrijevic and Saric [163] developed another transformation that decreased the size of the corresponding TSP. Behzad and Modarres [40] provided an efficient transformation. The main idea is simple. The number of waypoints remains unchanged and the waypoints in each cluster are connected in a cycle with all edges having zero cost. To avoid splitting the cycle the weights of inter-custer edges are increased by a large value M so that the solution cost rises by exactly kM. One obvious difference of our depth-first search approaches and solving the transformation for the GTSP is that we generate tours of length k, while in the compilation the tours have length n.

Approaches to find waypoints [64] often start with a collection of points that is not minimal but is easy to approximate and is expected to provide good coverage. The work in [668, 449] uses the skeleton of a 3D region, whose boundaries are defined by meshes, to define an initial set of waypoints, which is then pruned using integer-linear programming. Another approach selects waypoints from skeleton points of a 2D environment, but prunes the number of waypoints less aggressively with a simple heuristic since their goal is to keep visibility on a moving target while minimizing the motion of the robot, rather than inspecting the entire workspace [23]. The work in [394] uses a decomposition of a 2D environment into convex polygons to quickly place the waypoints without attempting to minimize their number. Medial axis points are also considered good waypoint candidates, but medial axis itself is hard to compute exactly, so approximation algorithms based on Voronoi diagrams have been explored [246].

There are different approaches to solving TSPs depending on the size of the problem. Branch-and-bound techniques that rely on sophisticated lower bounds have proven effective in solving large unconstrained problems [670]. Integer programming branch-and-cut solvers have been shown to be most effective in finding optimal solutions for highly constrained TSPs. Neighborhood local search algorithms are often used when the objective is to find satisfying tours [43].

Visiting several polygonal regions through a shortest tour has been tackled in [247] through self-organizing maps; a watchman route problem is handled by first providing a convex cover set of the environment. The work in [146] selects waypoints by creating a probabilistic roadmap [393] and then uses an approximate TSP solver on the shortest-path graph to compute a tour.

Sampling-based path planning is also used to iteratively construct a set of waypoints for a 3D environment and create a cyclic path containing them, which is then smoothed and shortened using heuristics so as to not reduce the space visible from the waypoints. The work in [503] uses a sampling-based approach to integrate waypoint selection and pathfinding, so as to guarantee waypoints can be reached given the robot's dynamics. These sampling-based approaches are shown to be probabilistically complete and to asymptotically converge to the optimal path.

The linear-time grass-firing algorithm has been proposed by [58]. Depth-first branch-and-bound goes way back to computer science history, we suggest consulting Korf's work [413].

Chapter 24
Industrial Production

A discrete event system (DES) is a dynamic system with discrete states and transitions, which are triggered by events. We apply a software model checker to a discrete event system that controls the industrial production of autonomous products. The flow of material is asynchronous and buffered. This chapter aims to find concurrent plans that optimize the throughput of the system. In mapping the discrete event system directly to the model checker, we model the production line as a set of communicating processes, with the movement of items modeled as channels. Experiments show that the model checker can analyze the DES, subject to the partial ordering of the product parts. It derives valid and optimized plans with several thousands of steps using constrained branch-and-bound.

24.1 Introduction

Discrete[1] *event (dynamic) systems* (DES) provide a general framework for systems where the system dynamics not only follow physical laws but also additional firing conditions. DES research is concerned about performance analysis, evaluation, and optimization of DES. As the systems are often only available as computer programs, it turns out to be difficult to describe the dynamics of these systems using closed-form equations. In many cases, *discrete event system simulation* (DESS) is chosen to describe the DES dynamics and for performance evaluation. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next. Each simulation activity is modeled by a process. The idea of a process is like the notion in model checking, and indeed one could write process-oriented simulations using independent processes. Most DESS systems store information about pending events in a data structure known as an *event queue*. Each item in the queue would at minimum contain the following information: a timestamp and a piece of software for executing an event. The typical operations on an event queue are inserting a new event and removing the next event (the one with the lowest timestamp) from the queue. It may also be necessary to cancel a scheduled event.

DESS is probably the most widely used simulation technique. Similar approaches are *system dynamics* (SD), and *agent-based simulation* (ABS). As the name suggests DES model a process as a series of discrete events. They are built using *entities* (objects that move through the system); *events* (processes which the entities pass through); and *resources* (objects which are needed to trigger event). SD is related to DES, focusing on *flows* around networks rather than *queueing systems*; it considers *stocks* (basic stores of objects); *flows* (movement of objects between different stocks in the system); *delays* (between the measuring and then acting on that measurement). ABS is a relatively new technique in OR and consists of: *autonomous agents* (self-directed objects which move about the system) and *rules* (which the agents follow to achieve their objectives). Agents move about the system interacting with each other and the environment. Earlier simulation software was efficient

[1] This chapter is based on joint work with Christoph Greulich. It puts together and improves the work from [201, 199, 304].

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_24

but platform-dependent, due to the need for stack manipulation. Modern software systems, however, support lightweight processes or threads. By the growing amount of non-determinism, however, DESS encounters its limits to optimize the concurrent acting of individual processes.

With the advances in technology, more and more complex systems were built, e.g., transportation networks, communication and computer networks, manufacturing lines. In these systems, the main dynamic mechanism in task succession stems from synchronization and competition in the use of common resources, which requires a policy to arbitrate conflicts and define priorities, all kinds of problems generally referred to under the generic terminology of *scheduling*. This type of dynamics hardly can be captured by differential equations or by their discrete-time analogues. This is certainly the reason why those systems, which are nevertheless true dynamic systems, have long been disregarded by formal method experts and have been rather considered by operations researchers and specialists of manufacturing with no strong connections with system theory. The dynamics are made up of *events*, which may have a continuous evolution imposed by some called software once they start, but this is not what one is mainly interested in: the primary focus is on the beginning and the end of such events, since ends can cause new beginnings. Hence, the word *discrete* includes *time* and *state*.

In this chapter, we utilize the state-of-the-art model checker Spin as a performance analysis and optimization tool together with its input language Promela to express the flow production of goods. There are several twists needed to adapt Spin to the optimization of DES(S) that are uncovered in the sequel of the text. Our running case study is the *Z2*, a physical monorail system for the assembling of taillights. Unlike most production systems, the Z2 employs agent technology to represent autonomous products and assembly stations. The techniques developed, however, will be applicable to most flow production systems. We formalize the production floor as a system of communicating processes and apply *Spin* for analyzing its behavior. Using optimization mechanisms implemented on top of Spin, in addition to the verification of the correctness of the model, we exploit its exploration process for optimization of the production.

For the *optimization via model checking*, we use many new language features from the latest version of the Spin model checker, including loops and native C-code verification.

24.2 Preliminaries

24.2.1 Discrete Event Simulation

An *entity* is an object of interest in the system, and an *attribute* is a (relevant) property of an entity. Attributes are *state variables*, while *activities* form part of the *model specification* and *delays* form part of the simulation result. The *(system) state* is a set of variables needed to describe the status of the system (e.g., length of a queue), which is aimed to be complete and minimal at any point in time.

The occurrence of a *primary event* (e.g., arrival) is scheduled at a certain time, while a *secondary event* (e.g., queueing) is triggered by a certain condition becoming true. An *event* is an occurrence which is instantaneous and may change the state of the system. The *(future) event list PQ* controls the simulation: it contains all future events that are scheduled and is ordered by increasing time of events. Operations on the *PQ* are *insert* an event into *PQ* (at an appropriate position!), *remove* first event from *PQ* for processing, and *delete* an event from *PQ*. Thus, *PQ* is a priority queue. As operations must be performed efficiently, the common implementation of an event queue is a (binary) *heap*. With such a data structure, access to the next event requires $O(1)$ time, while inserting/deleting an event requires $O(\log(n))$ time, where n is the number of events currently in the queue. Depending on the implementation (e.g., Fibonacci heaps), there are other trade-offs, with constant-time insertion and $O(\log(n))$ (amortized) deletion. The generic DES simulation algorithm looks as follows: 1) if *PQ* empty, then exit; 2) remove & process first primary event e from *PQ*; 3) if conditional event e' enabled, then remove & process e', goto 3, else goto 1.

Figure 24.1: Train game.

Program 24.1: Train game (initialization).

```
public class Event {
  public int arrival, type, number, location;
  public Event(int a, int t, int n, int l) { arrival = a; type = t; number = n; location = l; }
};
public class MainMenuController : MonoBehaviour {
  public static int [,] network = new int [,] {
    {0,100,200,-1,-1,-1,-1,-1,-1,-1},
    {100,0,-1,-1,-1,200,400,-1,-1,250},
    {200,-1,0,-1,-1,-1,100,-1,-1,-1},
    {-1,-1,-1,0,150,200,100,60,-1,200},
    {-1,-1,-1,150,0,350,-1,100,250,300},
    {-1,200,-1,200,350,0,-1,350,-1,100},
    {-1,400,100,100,-1,-1,0,-1,250,-1},
    {-1,-1,-1,60,100,350,-1,0,150,-1},
    {-1,-1,-1,-1,250,-1,250,150,0,-1},
    {-1,250,-1,200,300,100,-1,-1,-1,0}};
  public static string [] location = new string [] {"Bremen", "Hamburg", "Dortmund",
    "Frankfurt", "Munich", "Berlin", "Cologne", "Stuttgart", "Freiburg", "Dresden"};
  public static bool selected = false;
  public static int time = 0,loc = 0,number = 90,bonus = 0,kind = 0,balance = 200000;
  public static string dest = "",board = "";
  public static string [] type = new string [] {"ICE","IC","RE","EC","EN","RB"};
  public static int [] speed = new int [] {5,4,2,3,2,1};
  public static int [] capacity = new int [] {20,13,7,10,6,4};
  public static int [] cost = new int [] {1000000,250000,100000,200000,90000, 75000 };
  public static int [] trains = new int [] {0,0,0,0,0,0};
  public static int [] from = new int [] {5,9,2,6,8,7,6,8,5,0,5,9,2,1,4,7,6,8,5,0};
  public static int [] destination = new int [] {7,6,8,5,0,5,9,2,6,8,7,6,8,5,0,1,2,4,3,1};
  public static int [] status = new int[] {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  public static int [] train = new int[] {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  public static int [] types = new int[] {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  public static int [] duration = new int []
      {180,270,250,100,100,250,260,220,240,120,530,320,260,250,330,440,550,500,120,610};
  public static int [] reward = new int [] {8000,7000,5000,9000,10000,
      5000,6000,12000,4000,2000,3000,2000,6000,5000,3000,4000,5000,20000,2000,1000};
```

We assume exact timing, i.e., deterministic time. However, by different choices of points for generating successor events, the simulated DES itself is non-deterministic. Events inserted with priority t are generally assumed to remain unchanged until deletion at time t.

As an example of a simple discrete event system, we consider the *Train Game* (see Figure 24.1 and Programs 24.1 to 24.3). The story line of the game is the following. You are a railway company. You can buy trains to transport passengers. Every train type has its own cost and speed. Passengers impose start and target locations and max. travel time. For their successful transport a reward is paid. Trains operate between cities to be selected. Events happen chronologically. Instead of maintaining a priority queue we resort the events.

Program 24.2: Train game (event list, start and update function).

```
public static List<Event> events = new List<Event>();
public Text mapText, taskText, topText;
void Start () {
  try { mapText = GameObject.FindGameObjectWithTag("MapText").GetComponent<Text>(); }
  catch (Exception) { }
  try { taskText = GameObject.FindGameObjectWithTag("TaskText").GetComponent<Text>(); }
  catch (Exception) { }
  try { topText = GameObject.FindGameObjectWithTag("TopText").GetComponent<Text>(); }
  catch (Exception) { }
}
void Update () {
  if (topText != null) { topText.text = "Time: " + time + "Money: " + balance + "Location: " + location[loc]; }
  if (mapText != null) {
    mapText.text = "Map:\n";
    for (int i=0;i<location.Length;i++)
      for (int j=i+1;j<location.Length;j++)
        if (network[i,j] > 0)
          mapText.text += "" + location[i] + "->" + location[j] + "=" + network[i,j] +"\n";
  }
  if (taskText != null) {
    taskText.text = "Tasks:\n";
    for (int i=0;i<from.Length;i++)
      if (status[i] == 1)
        taskText.text += "->" + location[destination[i]] + "in " + duration[i] + "m, +" + reward[i] + "[" +
            train[i] + "]\n";
    for (int i=0;i<from.Length;i++)
      if (status[i] == 0)
        taskText.text += location[from[i]] + "->" + location[destination[i]] + "in " + duration[i] + "m, +" +
            reward[i] + "\n";
  }
}
```

24.2.2 Flow Manufacturing

Flow manufacturing systems are DES installed for products that are produced in high quantities. By optimizing the flow of production, manufacturers hope to speed up production at a lower cost, and in a more environmentally sound way. In manufacturing practice there are not only flow lines (with stations arranged one behind the other), but also more complex networks of stations at which assembly operations are performed (assembly lines). The considerable difference from flow lines, which can be analyzed by known methods, is that several required components are brought together to form a single unit for further processing at the assembly stations. An assembly operation can begin only if all required parts are available.

Performance analysis of flow manufacturing systems is generally needed during the planning phase regarding the system design, when the decision for a concrete configuration of such a system must be made. The planning problem arises, e.g., with the introduction of a new model or the installation of a new manufacturing plant. Because of the investments involved, an optimization problem arises. The expenditure for new machines, for buffer or handling equipment, and the holding costs for the expected work-in-process face revenues from sold products. The performance of a concrete configuration is characterized by the throughput, i.e., the number of items that are produced per time unit. Other performance measures are the expected work in process or the idle times of machines or workers.

We consider *assembly-line networks with stations*, which are represented as a directed graph. Between any two successive nodes in the network, we assume a buffer of finite capacity. In the buffers between stations and other network elements, work pieces are stored, waiting for service. At assembly stations, service is given to work pieces. Travel time is measured, and overall time is to be optimized.

In a general notation of flow manufacturing, system progress is non-deterministic and asynchronous, while the progress of time is monitored.

A *flow manufacturing system* is a tuple $F = (A, E, G, \prec, S, Q)$ where A is a set of all possible assembling actions; P is a set of n products; each $P_i \in P$, $i \in \{1, \ldots, n\}$, is a set of assembling actions, i.e., $P_i \subseteq A$; $G = (V, E, w, s, t)$ is a graph with start node s, goal node t, and weight function $w : E \to \mathbb{R}_{\geq 0}$; $\prec = (\prec_1, \ldots, \prec_n)$ is a vector of

Program 24.3: Train game: Starting a game and new arrivals.

```
  public void NewGame() {
    if (events.Count==0) return;
    selected = false;
    int differ = -time;
    events.RemoveAt(0);
    events.Sort((x, y) => x.arrival.CompareTo(y.arrival));
    Event e = events[0];
    int l = e.location, a = e.arrival, t = e.type, n = e.number;
    differ += a;
    for (int i = 0; i<reward.Length;i++)
     if (status[i] > 0) duration[i] -= differ;
    time = a; loc = l; number = n; kind = t;
  }
  public void StartTutorial() { SceneManager.LoadScene("Tutorial"); }
  public void StartMap() { SceneManager.LoadScene("Map"); }
  public void StartTasks() { SceneManager.LoadScene("Tasks"); }
  public void StartBuy() { SceneManager.LoadScene("Select"); }
  public void StartBoard() { if (events.Count==0) return; board = ""; SceneManager.LoadScene("Passenger"); }
  public void StartArrival() {
    if (selected) return;
    if (events.Count==0) return;
    for (int i=0;i<reward.Length;i++)
      if (destination[i] == loc && status[i] == 1 && train[i] == number) {
        bonus = reward[i];
        if (duration[i] > 0) balance += bonus;
        status[i] = 2;
      }
    dest = "";
    bool goal = true;
    for (int i=0;i<reward.Length;i++)
      if (status[i] < 2) goal = false;
    if (goal) SceneManager.LoadScene("Game"); else SceneManager.LoadScene("Arrival");
  }
  public void ShowCredits() { SceneManager.LoadScene("Credits"); }
  public void Exit() { Application.Quit(); }
}
```

assembling plans with each $\prec_i \subseteq A \times A$, $i \in \{1,\ldots,n\}$, being a partial order; $S \subseteq E$ is the set of assembling stations induced by a labeling $\rho : E \to A \cup \{\emptyset\}$, i.e., $S = \{e \in E \mid \rho(e) \neq \emptyset\}$; Q is a set of (FIFO) queues of finite size, i.e., $\forall q \in Q : |q| < \infty$, together with a labeling $\psi : E \to Q$;

Products P_i, $i \in \{1,\ldots,n\}$, travel through the network G, meeting their assembling plans in the form of orders $\prec_i \subseteq A \times A$ of the according assembling actions A. For defining the cost function we use the set of predecessor edges $Pred(e) = \{e' = (u,v) \in E \mid e = (v,w)\}$.

Let $F = (A, E, G, \prec, S, Q)$ be a flow manufacturing system. A *run* π is a schedule of triples (e_j, t_j, l_j) of edges e_j, queue insertion positions l_j, and execution time-stamps t_j, $j \in \{1,\ldots,m\}$. The set of all runs is denoted as Π. Each run π partitions into a set of *n plans* $\pi_i = (e_1,t_1,l_1),\ldots,(e_m,t_m,l_m)$, one for each product P_i, $i \in \{1,\ldots,n\}$. Each plan π_i corresponds to a *path*, starting at the initial node s and terminating at goal node t in G.

The *objective* in a flow manufacturing system for product i is to minimize

$$\max_{1 \leq i \leq n} wait(\pi_i) + time(\pi_i)$$

over all possible paths with initial node s and goal node t, where

- $time(\pi_i)$ is the *travel time* of product P_i, defined as the sum of edge costs $time(\pi_i) = \sum_{e \in \pi_i} w(e)$, and
- $wait(\pi_i)$ the *waiting time*, defined as $wait(\pi_i) = \sum_{(e,t,l),(e',t',l') \in \pi_i, e' \in Pred(e)} t - (t' + w(e'))$.

With $cost(\pi_i) = wait(\pi_i) + time(\pi_i)$, as *overall objective function* in a flow manufacturing system for product i we have

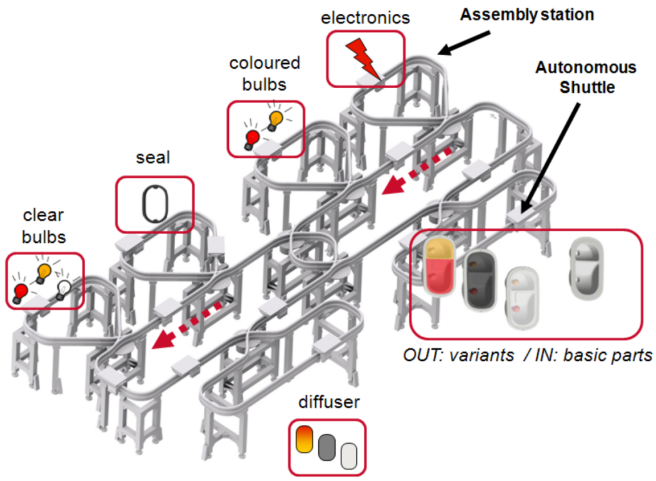


Figure 24.2: Assembly scenario for taillights.

$$\min_{\pi\in\Pi}\max_{1\le i\le n} cost(\pi_i) = \min_{\pi\in\Pi}\max_{1\le i\le n}\sum_{e\in\pi_i} w(e) + \sum_{(e,t,l),(e',t',l')\in\pi_i,e'\in Pred(e)} t-(t'+w(e'))$$
$$= \min_{\pi\in\Pi}\max_{1\le i\le n,(e,t,l)\in\pi_i} t+w(e)$$

subject to the side constraints that: time stamps on all runs $\pi_i = (e_1,t_1,l_1)\dots(e_m,t_m,l_m)$, $i\in\{1,\dots,n\}$ are monotonically increasing, i.e., $t_l \le t_k$ for all $1\le l<k\le m$; after assembling all products are complete, i.e., all assembling actions have been executed, so that for all $i\in\{1,\dots,n\}$ we have $P_i = \cup_{(e_j,t_j,l_j)\in\pi_i}\{\rho(e_j)\}$; the order of assembling product P_i on path $\pi_i = (e_1,t_1,l_1)\dots(e_m,t_m,l_m)$, $i\in\{1,\dots,n\}$, is preserved, i.e., for all $(a,a')\in\prec_i$ and $a=\rho(e_j), a'=\rho(e_k)$ we have $j<k$; all insertions to queues respect their sizes, i.e., for all $\pi_i = (e_1,t_1,l_1)\dots(e_m,t_m,l_m)$, $i\in\{1,\dots,n\}$, we have that $0\le l_j < |\psi(e_j)|$.

24.3 Case Study

The Z2 production floor unit consists of six workstations where human workers assemble parts of automotive taillights. The system allows production of certain product variations and reacts dynamically to any change in the current order situation, e.g., a decrease or an increase in the number of orders of a certain variant. As individual production steps are performed at the different stations, all stations are interconnected by a monorail transport system. The structure of the transport system is shown in Figure 24.2. On the rails, autonomously moving shuttles carry the products from one station to another, depending on the products' requirements. The monorail system has multiple switches which allow the shuttles to enter, leave or pass workstations and the central hubs. The goods transported by the shuttles are also autonomous, which means that each product decides on its own which variant to become and which station to visit. This way, a decentralized control of the production system is possible.

The modular system consists of six different workstations, each operated manually by a human worker and dedicated to one specific production step. Different parts can be used to assemble different variants of the taillights. At the first station, the basic metal-cast parts enter the monorail on a dedicated shuttle. The monorail connects all stations; each station is assigned to one specific task, such as adding bulbs or electronics. Each taillight is transported from station to station until it is assembled completely.

In the DESS implementation of the Z2 system (see Figure 24.3), every assembly station, every monorail shuttle and every product is represented by a software process. Even the RFID readers which keep track of product positions are represented by software processes, which decide when a shuttle may pass or stop.

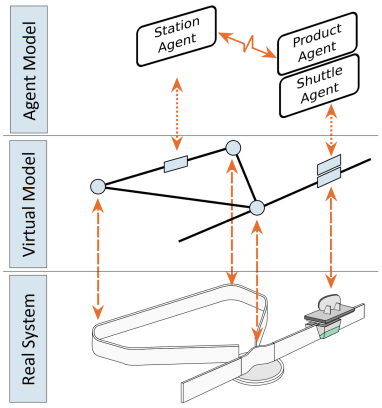


Figure 24.3: Layered architecture of models and agents for the Z2.

Most processes in this DESS resemble simple reflex methods. These processes just react to requests or events which were caused by other processes or the human workers involved in the manufacturing process. In contrast, the processes which represent products are actively working towards their individual goal of becoming a complete taillight and reaching the storage station. To complete its task, each product has to reach sub-goals which may change during production as the order situation may change. The number of possible actions is limited by sub-goals which have already been reached, since every possible production step has individual preconditions.

The product processes constantly request updates regarding queue lengths at the various stations and the overall order situation. The information is used to compute the utility of the expected outcome of every action. High utility is given when an action leads to fulfillment of an outstanding order and takes as little time as possible. Time, in this case, is spent either on actions, such as moving along the railway or being processed, or on waiting in line at a station or a switch.

The Z2 DES was developed strictly for the purpose of controlling the Z2 monorail hardware setup. Nonetheless, due to its hardware abstraction layer, the Z2 DES can be adapted to other hardware or software environments. By replacing the hardware with other processes and adapting the monorail infrastructure into a directed graph, the Z2 DES has been compiled to a DESS. Such an environment, which treats the original Z2 modules like black boxes, can easily be hosted by a DESS. Experiments showed how closely the simulated and the real-world scenarios match.

For this study, the model with timers (to measure the time taken between two graph nodes) was provided. Since the hardware includes many RFID readers along the monorail, which all are represented by an agent and a node within the simulation, we simplified the graph and kept only three types of nodes: switches, production station entrances and production station exits. The resulting abstract model of the system is a weighted graph (see Figure 24.4), where the weight of an edge denotes the traveling/processing time of the shuttle between two respective nodes.

24.4 Promela Specification

Promela is the input language of the model checker Spin, the ACM-awarded popular open-source software verification tool, designed for the formal verification of multi-threaded software applications, and used by thousands of people worldwide. Promela defines asynchronously running communicating processes, which are compiled to finite state machines. It has a C-like syntax and supports bounded channels for sending and receiving messages. Channels in Promela follow the FIFO principle: They implicitly maintain order of incoming messages and can be limited to a certain buffer size. Consequently, we are able to map edges to communication channels. Unlike the original Z2 ABS, the products are not considered to be decision making entities within our Promela

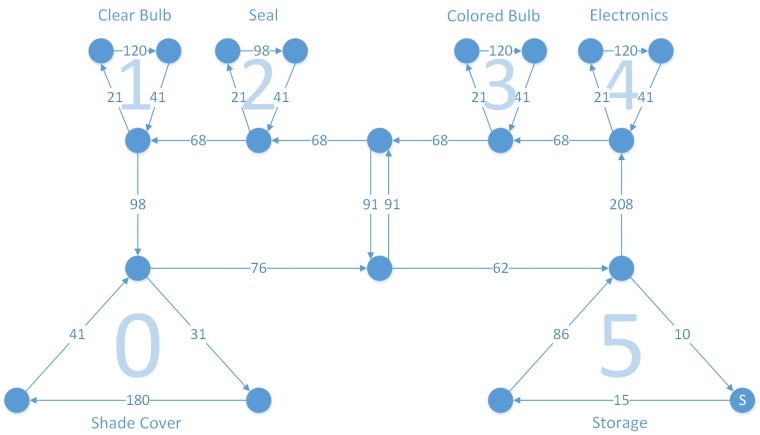


Figure 24.4: Weighted graph model of the assembly scenario.

model. Instead, the products are represented by messages which are passed along the *node processes*, which resemble switches, station entrances and exits.

Unlike the original DESS, the Promela model is designed to apply a branch-and-bound optimization to evaluate the optimal throughput of the original system. Instead of local decision making, the various processes have certain nondeterministic options for handling incoming messages, each leading to a different system state. The model checker systematically computes these states and memorizes paths to desirable outcomes when it ends up in a final state. As mentioned before, decreasing production time for a given number of products increases the utility of the final state.

We derive a Promela model of the Z2 as follows. First, we define a global bound for the number of stations and switches. We also define the data type storing the index of the shuttle/product to be `byte`. In the model, switches are realized as processes and edges between the units by the following channels.

```
chan entrance_to_exit[STATIONS] = [1] of {shuttle};
chan exit_to_switch[STATIONS] = [BUFFERSIZE] of {shuttle};
chan switch_to_switch[SWITCHES] = [BUFFERSIZE] of {shuttle};
chan switch_to_entrance[STATIONS] = [BUFFERSIZE] of {shuttle};
```

As global variables, we have bitvectors for marking the different assemblies.

```
bit metalcast[SHUTTLES]; bit electronics[SHUTTLES];
bit bulb[SHUTTLES]; bit seal[SHUTTLES]; bit cover[SHUTTLES];
```

Additionally, a bitvector denotes when a shuttle with a fully assembled item has finally arrived at its goal location. Another bitvector sets for each shuttle whether it has to acquire a colored or a clear bulb.

```
bit goals[SHUTTLES]; bit color[SHUTTLES];
```

A switch is a process that controls the flow of the shuttles. In the model, a non-deterministic choice is added either to enter the station or to continue traveling onwards on the cycle. Three of four switching options are made available, as immediately re-entering a station from its exit is prohibited.

```
proctype Switch(byte in; byte out; byte station) {
  shuttle s;
  do
  :: exit_to_switch[station]?s; switch_to_switch[out]!s;
  :: switch_to_switch[in]?s; switch_to_switch[out]!s;
  :: switch_to_switch[in]?s; switch_to_entrance[station]!s;
  od
}
```

The entrance of a manufacturing station takes the item from the according switch and moves it to the exit. It also controls that the manufacturing complies with the capability of the station.

First, the assembling of product parts is different at each station; in the stations 1 and 3 we have the insertion of bulbs (station 1 provides colored bulbs, station 3 provides clear bulbs), station 2 assembles the seal, station 4 the electronics and station 0 the cover. Station 5 is the storage station where empty metal casts are placed on the monorail shuttles and finished products are removed to be taken into storage. Secondly, there is a partial order of the respective product parts to allow flexible processing and a better optimization based on the current load of the ongoing production.

```
proctype Entrance(byte station) {
  shuttle s;
  do
  :: switch_to_entrance[station]?s;
     entrance_to_exit[station]!s
     if
     :: (station == 4) -> electronics[s] = 1;
     :: (station == 3 && !color[s]) -> bulb[s] = 1;
     :: (station == 2)-> seal[s] = 1;
     :: (station == 1 && color[s]) -> bulb[s] = 1;
     :: (station == 0 && seal[s] && bulb[s] && electronics[s])-> cover[s] = 1;
     :: (station == 5 && cover[s]) -> goals[s] = 1;
     :: else
     fi
  od
}
```

An exit is a node that is located at the end of a station, at which assembling took place. It is connected to the entrance of the station and the switch linked to it.

```
proctype Exit(byte station) {
  shuttle s;
  do
  :: entrance_to_exit[station]?s; exit_to_switch[station]!s;
  od
}
```

A *hub* is a switch that is not connected to a station but provides a shortcut in the monorail network. Again, three of four possible shuttle movement options are provided

```
proctype Hub(byte in1; byte out1; byte in2; byte out2) {
  shuttle s;
  do
  :: switch_to_switch[in1]?s; switch_to_switch[out1]!s;
  :: switch_to_switch[in1]?s; switch_to_switch[out2]!s;
  :: switch_to_switch[in2]?s; switch_to_switch[out1]!s;
  od
}
```

In the initial state, we start the individual processes, which represent switches. This defines the network of the monorail system. Moreover, initially, we have that the metal cast of each product is already present on its carrier, the shuttle. The coloring of the taillights can be defined at the beginning or in the progress of the production. Last, but not least, we kick off by inserting shuttles on the starting rail (at station 5).

```
init {
  atomic {
     byte i;
     c_code { cost = 0; }
     c_code { best_cost = infinity; }
     for (i : 0 .. (SHUTTLES)/2)){ color[i] = 1; }
     for (i : 0 .. (SHUTTLES-1)) { metalcast[i] = 1; }
     for (i : 0 .. (STATIONS-1)) { run Entrance(i); run Exit(i); }
     run Switch(7,0,5); run Switch(0,1,4);
     run Switch(1,2,3); run Switch(3,4,2);
     run Switch(4,5,1); run Switch(5,6,0);
     run Hub(2,3,8,9); run Hub(6,7,9,8);
```

Algorithm 24.1: DFBnB Algorithm.

DFBnB$(F = (A, E, G, \prec, S, Q))$
  Initialize upper bound U
  $\pi' \leftarrow \pi \leftarrow \emptyset$
  $DFS(F, (s, \ldots, s), 0, U)$
  return π'

DFS(F, u, π, U)
  $\pi \leftarrow extend(\pi, u)$
  if $(u = (t, \ldots, t))$
    if $(cost(\pi) < U)$ $\pi' \leftarrow \pi$; $U \leftarrow cost(\pi)$
  else for each v **in** $successors(u)$
      if $(cost(\pi) + h(v) < U)$ $DFS(F, v, \pi, U)$

```
    for (i : 0 .. (SHUTTLES-1)) { exit_to_switch[5]!i; }}
}
```

We also heavily made use of the term `atomic`, which enhances the exploration for the model checker, allowing it to merge states within the search. In contrast to the more aggressive `d_step` keyword, in an `atomic` block all communication queue actions are blocking.

24.5 Optimized Scheduling

We improve branch-and-bound (BnB) optimization. *Branching* is the process of spawning subproblems, while *bounding* refers to ignoring partial solutions that cannot be better than the current best solution. To this end, lower and upper bounds and are maintained as global control values on the solution quality, which improves over time.

For applying BnB to general flow manufacturing systems, we extend depth-first search (DFS) with upper (and lower) bounds. In this context, branching corresponds to the generation of successors, so that DFS can be casted as generating a *branch-and-bound search tree*. One way of obtaining a lower bound L for the problem state u is to apply an *admissible heuristic* h with $L(u) = g(u) + h(u)$, where g denotes the cost for reaching the current node from the root, and h is a function that always underestimates the remaining cost to reach a goal.

As with standard DFS, the first solution obtained might not be optimal. With *depth-first branch-and-bound* (DFBnB), however, the solution quality improves over time together with the global value U until eventually the lower bound $L(u)$ at some node u is equal to U (see Algorithm 24.1).

We applied branch-and-bound optimization within Spin. Essentially, the model checker can find traces of several hundreds of steps and provides trace optimization by finding the shortest path towards a counterexample if run with the parameter `./pan -i`. As these traces are step-optimized, and not cost-optimized, we use variable *cost* as follows.

```
c_state "int min_cost" "Hidden" c_state "int min_cost" "Hidden"
c_code { int cost; } c_code { int cost[SHUTTLES]; }
c_track "cost" "sizeof(int)" "Matched" c_track "cost" STRING "Matched"
```

While the *cost* variable increases the amount of memory required for each state, it also limits the power of Spin's built-in duplicate detection, as two otherwise identical states are considered different if reached by different accumulated cost. If the search space is small, so that it can be explored even for the enlarged state vector, then this option is sound and complete, and finally returns the optimal solution to the optimization problem. However, there might be simply too many repetitions in the model so that introducing cost to the state vector leads to a drastic increase in state space size, so that otherwise checkable instances now become intractable. We noticed that even by concentrating on safety properties (such as the failed assertion mentioned), the insertion of costs causes trouble.

24.5.1 Guarded Branching

Costs have to be tracked for every shuttle individually. The variable *cost* of the most expensive shuttle indicates the duration of the whole production process. Furthermore, the cost total provides insight regarding unnecessary detours or long waiting times. Hence, minimizing both criteria are the optimization goals of this model.

In Promela, every `do`-loop is allowed to contain an unlimited number of possible options for the model checker to choose from. The model checker randomly chooses between these options; however, it is possible to add an *if*-like condition to an option: If the first statement of a `do` option holds, Spin will start to execute the following statements, otherwise, it will pick a different option. Since the model checker explores any possible state of the system, many of these states are technically reachable but completely useless from an optimization point of view. To reduce the state space size to a manageable level, we add constraints to the relevant receiving options in the `do`-loops of every node process.

Peeking into the incoming queue to find out which shuttle is waiting to be received is already considered a complete statement in Promela. Therefore, we exploit C-expressions (`c_expr`) to combine several operations into one atomic statement. For every station t and every incoming channel q, a function `prerequisites`(t,q) determines if the first shuttle in q meets the prerequisites for t.

```
shuttle s;
do
:: c_expr{prerequisites(Px->q,Px->t)} -> channel[q]?s; channel[out]!;
od
```

At termination of a successful run, we use use the integer array `cost[SHUTTLES]` of the Promela model. It enables each process to keep track of its local cost vector and is increased by the cost of each action as soon as the action is executed. This enables the model checker to print values to the output only if the values of the current `max` cost and `total` cost have improved.

```
terminate:
c_code {
 int max = 0, total = 0, j;
 for (j=0; j<SHUTTLES; j++) {
   total += cost[j];
   if (cost[j] > max) max = cost[j]; }
 if (max < min_cost) { min_cost = max; putrail(); Nr_Trails--; };
}
```

For solution reconstruction, we write a file for each new cost value obtained, temporarily renaming the trail file as follows.

```
    char mytrailfile[512];
    sprintf(mytrailfile, "%s_t%d_st%d.pr", base,min_cost,total);
    char* y = mytrailfile;
    swap(&TrailFile, &y);
    putrail();
    swap(&y, &TrailFile);
```

24.5.2 Process Synchronization

Due to the nature of the state space search of the model checker, processes in the Promela model do not make decisions. Nonetheless, the given model is a distributed DES consisting of a varying number of processes, which potentially influence each other if executed in parallel.

We addressed this problem by introducing an event-based time progress to the Promela model. Whenever a shuttle s travels along one of the edges, the corresponding message is put into a channel and the cost of the respective shuttle is increased by the cost of the given edge.

```
shuttle s;
do
:: c_expr{ canreceive(channel,Px->q,Px->station) }
   -> channel[q]?s
   c_code { cost[s] += Px->c; }
   channel[out] ! s;
od
```

We introduce an atomic C function *canreceive*(q) that returns true, only if the first item s of q has minimal *cost(s)*, changing the receiving constraint to the following.

```
c_code {
int canreceive(int channeltype, int arrayidx, int station) {
  int channelidx = -1;
  switch(channeltype) {
        case xyz: channelidx = now.xyz[arrayidx]; break; [...]
  }
  if(channelidx > -1 && q_len(channelidx) > 0) {
    int shuttle = qrecv(channelidx, 0, 0, 0);
    int minimum = infinity;
    for (int j=0; j<SHUTTLES; j++) {
      if (cost[j] < minimum) minimum = cost[j]; }
    return (minimum == cost[shuttle]); }
  return 0;
}
```

Within Spin, the global Boolean variable `timeout` is automatically set to *true* when all current processes are unable to proceed, e.g., because they cannot receive a message. Consequently, for every shuttle p, all processes will be blocked, and `timeout` will be set to *true*. We add a process that enforces time progress whenever `timeout` occurs (*final* is a macro for reaching the goal).

```
active proctype watchdog() {
   do
   ::timeout -> c_code{ increase(); } ; assert(!final);
   od
}
```

Time is delayed as follows: if the minimum event in the future event list is blocked (e.g., a shuttle is not first in its queue), we compute the wake-up time of the second-best event. If the two are of the same time, a time increment of 1 is enforced. In the other case, the second-best event time is taken as the new one for the first. It is easy to see that this strategy eventually resolves all possible deadlocks. Its implementation is as follows.

```
int increase() {
  int j, l = 0, minimum = cost[0];
  for (j=1; j<SHUTTLES; j++)
    if (cost[j] < minimum) { minimum = cost[j]; l = j; }
  int second = infinity;
  for (j=0; j<SHUTTLES; j++) {
    if (cost[j] < second && cost[j] > minimum)
     second = cost[j]; }
  cost[l] = (second == infinity) ? minimum + 1 : second;
}
```

As a summary, the constrained and bounded depth-first exploration has turned into the automated generation of the underlying state space of the DES, using c-code to preserve the causality of actions and to simulate the future event list.

```
class Arena {
  public:
  Move rollout[MaxLength]; int length;
  Arena() {
    length = 0;
    for (int i = 0; i< STATIONS; i++) {
      switch2entrance[i]->clear(); exit2switch[i]->clear();
      entrance2exit[i]->clear();
    }
    for (int i = 0; i< STATIONS + 2*HUBS; i++) switch2switch[i]->clear();
    for (int i = 0; i < SHUTTLES; i++) {
      wait[i] = 0; cost[i] = i * 70; goals[i] = 0; color[i] = i%2;
      metalcast[i] = 1; diffusor[i] = electronics[i] = bulb[i] = seal[i] = 0;
      switch2entrance[5]->push(i);
    }
  }
  int code (Move m) { return m; }
  int legalMoves (Move moves [MaxLegalMoves]) {
    int m[3], mvs = 0;
    while (mvs == 0) {
      for (int p = 0; p < agent.size(); p++) {
        int k = agent[p]->nextLegalMove(m);
        for(int l=0;l<k;l++) moves[mvs++] = p*3 + m[l];
      }
      if (mvs == 0) increase_time();
    }
    return mvs;
  }
  void play (Move m) { rollout[length++] = m; agent[m/3]->executeMove(m%3); }
  bool terminal () {
    int reached = 1;
    for (int j=0; j<SHUTTLES; j++) reached &= goals[j];
    return (reached) || length == MaxLength-1;
  }
  double score () {
    int maximum = 0, total = 0;
    for (int j=0; j<SHUTTLES; j++) if (cost[j] > maximum) maximum = cost[j];
    int reached = 0;
    for (int j=0; j<SHUTTLES; j++) reached += !goals[j];
    return (reached * 1000) + maximum;
  }
}
```

Figure 24.5: Code for Z2 multiagent system optimization.

```
class Agent {
  public:
  Agent() {}
  virtual void executeMove(int m) = 0;
  virtual int nextLegalMove(int* moves) = 0;
};
```

Figure 24.6: Code for abstract agent class.

24.6 Game Encoding

In the encoding as a single-player game, the number of acting agents is significantly reduced in comparison to the original DES. Like the encoding of model-checker-based approaches, decision making is modeled into the nodes while shuttles are merely integer values which are passed along the edges.

Each edge is modeled as a queue to make sure that no shuttle can pass another. When put on an edge, a shuttle receives a waiting time which corresponds to the cost of the specific edge. A synchronizing function ensures that time progresses for all shuttles. The node at the end of a directed edge is allowed to receive a shuttle only if

```
class Switch : public Agent {
  public:
  int In, Out, Station, B, C;
  Switch(int in, int out, int s, int b, int c):
  Agent(),In(in),Out(out),Station(s),B(b),C(c) {}
  void executeMove(int move) {
    if (move == 0) {
      int Shuttle = switch2switch[In]->pop();
      wait[Shuttle] += C; cost[Shuttle] += C;
      switch2entrance[Station]->push(Shuttle);
    }
    if (move == 1) {
      int Shuttle = switch2switch[In]->pop();
      wait[Shuttle] += B; cost[Shuttle] += B;
      switch2switch[Out]->push(Shuttle);
    }
    if (move == 2) {
      int Shuttle = exit2switch[Station]->pop();
      wait[Shuttle] += B; cost[Shuttle] += B;
      switch2switch[Out]->push(Shuttle);
    }
  }
  int nextLegalMove(int* moves) {
    int mvs = 0;
    if (receives(SW2SW_EN,In,Station)) moves[mvs++] = 0;
    if (receives(SW2SW_PASS,In,Station)) moves[mvs++] = 1;
    if (receives(EX2SW, Station,Station)) moves[mvs++] = 2;
    return mvs;
  }
};
```

Figure 24.7: Code for one agent.

```
void increase_time() {
  int min = INF, d = 1;
  for (int p = 0; p < SHUTTLES; p++)
    if (0 < wait[p] && wait[p] < min) min = wait[p];
  if (min < INF) d = min;
  for (int p = 0; p < SHUTTLES; p++)
    if (wait[p] - d >= 0) { wait[p] -= d; cost[p] += d; } else wait[p] = 0;
}
bool receives(int channeltype, int i, int station) {
  int result = 0; Channel* channel = NULL;
  switch(channeltype) {
    case EN2EX: channel = entrance2exit[i]; break;
    case EX2SW: channel = exit2switch[i]; break;
    case SW2SW_PASS: channel = switch2switch[i]; break;
    case SW2SW_EN: channel = switch2switch[i]; break;
    case SW2EN:
      if (entrance2exit[station]->length() >= 1) channel = NULL;
      else channel = switch2entrance[station];
      break;
  }
  if (channel != NULL && channel->length() > 0) {
    int shuttle = channel->front();
    if (wait[shuttle] <= 0) result = 1;
  }
  return result;
}
```

Figure 24.8: Code for increase-time and receive action.

it is first in its queue and its waiting time has passed. If a shuttle can be received by a node, the node provides a legal move for each outgoing edge. Hence, a set of all legal moves over all active agents can be obtained.

To play the game, the player has to choose one of the agents and one of its actions as the next move. The goal of the game is to finish a predefined number of products in the shortest possible time before a predefined length is exceeded. The smaller the makespan for each agent found by the algorithm the higher the score of the play.

Table 24.1: Sequences of events for $n = 3$ products. (*Product* $\Rightarrow$ *Station*, where $\Rightarrow$ indicates a finished production step.)

ABS	LVT	DES
$0 \Rightarrow 4$	$0 \Rightarrow 4$	$0 \Rightarrow 4$
$1 \Rightarrow 2$	$1 \Rightarrow 4$	$1 \Rightarrow 4$
$0 \Rightarrow 3$	$2 \Rightarrow 4$	$2 \Rightarrow 4$
$2 \Rightarrow 1$	$0 \Rightarrow 3$	$0 \Rightarrow 3$
$0 \Rightarrow 2$	$2 \Rightarrow 3$	$1 \Rightarrow 2$
$1 \Rightarrow 4$	$1 \Rightarrow 2$	$2 \Rightarrow 3$
$0 \Rightarrow 0$	$1 \Rightarrow 1$	$0 \Rightarrow 2$
$2 \Rightarrow 4$	$2 \Rightarrow 2$	$1 \Rightarrow 1$
$0 \Rightarrow 5$	$1 \Rightarrow 0$	$2 \Rightarrow 2$
$1 \Rightarrow 1$	$0 \Rightarrow 2$	$0 \Rightarrow 0$
$2 \Rightarrow 2$	$2 \Rightarrow 0$	$1 \Rightarrow 0$
$1 \Rightarrow 0$	$0 \Rightarrow 0$	$2 \Rightarrow 0$
$2 \Rightarrow 0$	$1 \Rightarrow 5$	$0 \Rightarrow 5$
$1 \Rightarrow 5$	$2 \Rightarrow 5$	$1 \Rightarrow 5$
$2 \Rightarrow 5$	$0 \Rightarrow 5$	$2 \Rightarrow 5$

More formally, the (board) game is defined as (B, b_0, d, F, r) where B is the set of (board) positions, in our case consisting of all queue content, shuttle locations, and their respective cost values. The start position s_0 has all shuttles and all queues being empty; $d\colon B \to 2^B$ specifies the set of allowed actions for each $q \in B$, The set of final positions F consists of all states in which either all the individual goals or the maximal step sized is reached, and $r\colon B \to \mathbb{N}$ is the score function adding a constant (e.g., 1,000) for each individual unreached goal, on top of the maximum of the individual cost values.

The components of the game induce a tree in the natural way with B as nodes, root b_0, d as edges and the final positions as leaves. A play(out) is then a path in the tree from b_0 to some leaf.

The software implementation (see Figures 24.5 to 24.8) is based on a framework which allows to employ several search algorithms such as MCS, NMCS, NRPA, BEAM-NRPA and HD-NRPA. For our experiments, we only focused on HD-NRPA since it is the most advanced implementation and provided the best results.

24.7 Evaluation

We compare the results of the exploration minimizing local virtual time (LVT) to the ones simulating the discrete event system (DES). We also present results of simulation runs of the original implementation on hardware. The Promela models do not rely on local decision making but on searches for an optimal solution systematically. Therefore, both Promela models resemble a centralized planning approach.

For smaller problems we experimented with Spin's parallel BFS, as it computes optimal-length counterexamples. The hash table is shared based on compare-and-swap (CAS). We also tried supertrace (bitstate hashing) as a trade-off. Unfortunately, we had to drop the experiments for cost optimization. Swarm tree search found many solutions, but due to the increased amount of randomness, for the optimized scheduling in general no better results than ordinary DFS were found.

In each experiment run, a number of $n \in \{2, \ldots, 20\}$ shuttles carry products through the facility. All shuttles with even IDs acquire clear bulbs, all shuttles with odd IDs acquire colored ones.

A close look at the experiment results of every simulation run reveals that, given the same number of products to produce, all three approaches result in different sequences of events. However, LVT and DES propose the same sequence of production steps for the product of each shuttle. The example given in Figure 24.1 shows that for all shuttles $0 \ldots 2$, the scheduling sequence is the same in LVT and DES, while the original ABS often proposes a different schedule. In the given example, both LVT and DES propose a sequence of $4, 2, 1, 0, 5$ for shuttle 1. To the contrary, the ABS approach proposes $2, 1, 4, 0, 5$ for shuttle 1. The same phenomenon can be observed for every $n \in \{2, \ldots, 20\}$ number of shuttles.

All three simulation models keep track of the local production time of each shuttle's product. In ABS and LVT simulation, minimizing maximum local production time is the optimization goal. Steady, synchronized progress

Table 24.2: Simulated production times for n products in the original ABS and Spin simulation, including the amount of RAM required to compute the given result. (* indicates complete state space exploration within the given RAM usage.)

	ABS	LVT		DES		NRPA	
Products	Max. Prod. Time	Max. Prod. Time	RAM	Max. Prod. Time	RAM	Max. Prod. Time	Length
2	4:01	3:24	987 MB*	2:53	731 MB*	2:54	48
3	4:06	3:34	2154 MB*	3:04	503 MB	2:59	72
4	4:46	3:56	557 MB	3:13	519 MB	3:08	99
5	4:16	4:31	587 MB	3:25	541 MB	3:13	123
6	5:29	4:31	611 MB	3:34	565 MB	3:22	153
7	5:18	5:08	636 MB	3:45	587 MB	3:38	186
8	5:57	5:43	670 MB	3:55	610 MB	3:45	213
9	6:00	5:43	692 MB	4:06	635 MB	3:52	240
10	6:08	5:43	715 MB	4:15	557 MB	3:52	267
20	9:03	8:56	977 MB	5:59	857 MB	5:40	516

of time is maintained centrally after every production step. Hence, whenever a shuttle has to wait in a queue, its total production time increases. For the DES model, progress of time is managed differently. Results show that max. production time in DES is lower than for LVT and ABS in all cases.

For every experiment, the amount of RAM required by DES to determine an optimal solution is slightly lower than the amount required by LVT, as shown in Table 24.2. While the LVT required several iterations to find an optimal solution, the first valid solution found by DES was already the optimal solution in any conducted experiment. However, the LVT model can search the whole state space within the RAM limit (given by our machine) for $n \leq 3$ shuttles, whereas the DES model is unable to search the whole state space for $n > 2$. For every experiment with $n > 3$ (LVT) or $n > 2$ (DES) shuttles respectively, searching the state space for better results was cancelled when the RAM limit was reached.

While the experiments indicate that the DES is faster and more memory efficient than the LVT approach, we observe that the mapping cost to time in the DES is limited. Assuming that events are processed by the time stamp while inserted in the priority queue, is a limitation. We look at extensions of the future event list supporting the priority queue operation *increaseKey*. In our experiment if one element in a process queue was delayed, all the ones behind it were delayed as well. While DES and LVT are both sound in resolving deadlocks, LVT has the more accurate representation for the progress of time.

As with the DES/LVT model, in cost we measure travel time plus an initial waiting time. To assist the solver in finding valid solutions, we added the objective function to the term $(e_r * 10) + (b_r * 10) + (s_r * 10) + (d_r * 100)$, where e_r, b_r, s_r, and d_r are the violations to the assembling status of electronics, bulbs, seals, and diffusors, respectively.

24.8 Summary

Simulation provides a method to approximate the behaviour in a real system (and, hence, can be used for testing scenarios). Constructing the model can prove useful in achieving greater understanding of the system. In this chapter, we presented a novel approach for model checking (instead of simulating) DES. The research is motivated by our interest in finding and comparing centralized and distributed solutions to the optimization problems in autonomous manufacturing.

Using model checking for optimizing DES is a relative new playground for formal method tools in the form of a new analysis paradigm. The formal model in Promela reflects the routing and scheduling of entities in the DES. Switches of the rail network were modeled as processes, the edges between the switches as communication channels. Additional constraints to the order of production steps enable us to carry out a complex planning and scheduling task. There is lot of room for improvement in the decentralized solution, since the model checker found more efficient ways to route and schedule the shuttles on several occasions. Furthermore, the model checker could derive optimized plans of several thousand steps. As with directed model checking, heuristics are expected to guide the search towards finding a good schedule faster. By looking at the limits and possibilities of LVT and DES, alternatives to represent time have to be considered.

Randomized Monte-Carlo search performed even better than a systematic exploration. The advanced NRPA algorithm learns good successors and has a smaller memory footprint than exhaustive methods (if they store states for eliminating duplicates). The main advantage with respect to a systematic enumeration of the state space is that the random choices do not rely on a fixed traversal ordering, and that for a good performance, we do not have to assist the model checker by ordering the non-deterministic choices manually.

Compared to depth-first search and branch-and-bound, Monte-Carlo tree search is less dependent on finding a very good ordering of the successors. Even hand-coded pruning like checking prerequisites was not needed to arrive at high-quality solutions in a short amount of time. Nonetheless, there is noticeable impact of hand-coded information, as additional guidance information encoded in the cost function increased the performance of the search substantially.

So far, we haven't extended Spin by Monte-Carlo tree search but gave an alternative implementation based on an existing single-player game optimization framework. Thus, on a first glance, a head-to-head comparison is seemingly unfair: Spin is aimed at general software verification, and with a granularity on lines of the running Promela code, it does produce much longer traces than the search framework, which is based on non-deterministic action choices only. Moreover, Spin already supports randomization in its swarm search wrapper, but swarm algorithms are based on random depth-first search, while Monte-Carlo search with policy adaptation improves over time, and, thus, learns the structure of the underlying problem. Nestedness of the search leads to exponential refreshment of policies, and, therefore, offers a trade-off between exploitation and exploration.

The interface for such randomized search with Monte-Carlo tree search is simple and flexible, only the functions rollout and adaptation have to be implemented properly. In fact, we are confident that the implementation of a Model Checker based on Monte-Carlo tree search is only a matter of time. At the end, only the successor generating function must be modified. This line of thinking could be viewed as putting together the power of directed model checking, guiding the search space traversal, and evolutionary algorithms, maintaining a population of policies based on random runs.

24.9 Bibliographic Notes

The model checker Spin has been implemented by Gerard Holzmann [347]. The optimization approach originally invented for Spin was designed for state space trees [548, 547], while the proposed approach also supports state space graphs. Scheduling via model checking has been pioneered by Binksma [83], and Wijs [654]. The work is especially inspired by [83, 454] and [547]. In standard Spin, the trivial heuristic $h \equiv 0$ is used, but in HSF-Spin [215], a few heuristic functions have been implemented. Bošnački and Dams [65] suggested to exploit the *timeout* command to model real-time with Spin.

One of the most interesting problems in manufacturing is *job shop scheduling* [30]. When solving the scheduling problem, a set of n jobs must assigned to a set of m machines. Consequently, the total number of possible solutions is $(n!)^m$. The problem complexity grows when the number of required resources increases, e.g., by adding specific tools or operators to run machines. For an additional set k of necessary resources, the number of possible solution increases to $((n!)^m)^k$ [573]. In the related *flow shop scheduling* problem, a fixed sequence of tasks forms a job [276]. It is applicable to optimize the *makespan*. The real-world *Z2* production floor unit has been presented in [480]. The Z2 DES has been transferred to a DESS in [305] and a centralized solution minimizing the agents' local virtual time has been given by [303].

Flow line analysis is a more complex setting, often done with queuing theory [463, 93]. Pioneering work in analyzing assembly queuing systems with synchronization constraints analyzes assembly-like queues with unlimited buffer capacities [322]. It shows that the time an item must wait for synchronization may grow without bound, while limitation of the number of items in the system works as a control mechanism and ensures stability. Work on assembly-like queues with finite buffers all assume exponential service times [48, 453, 348]. A rare example of model checking flow production applied timed automata that were used for simulating material flow in agricultural production [330].

The automated generation of plans for a given task is an integral part of problem solving in a computer. In *action planning* [491], we are confronted with the descriptions of the initial state, the goal (states) and the available actions. Based on these we want to find a plan containing as few actions as possible (in case of no- or unit-cost actions) or with the lowest possible total cost (in case of general action costs). The process of fully-automated property validation and correctness verification is referred to as *model checking* [129]. Given a formal model of a system M and a property specification ϕ in some form of temporal logic like LTL [290], the task is to validate, whether or not the specification is satisfied in the model, $M \models \phi$. If not, a model checker usually returns a counterexample trace as a witness for the falsification of the property.

Planning and model checking have much in common [295, 126]. Both rely on the exploration of a potentially large state space of system states. Usually, model checkers only search for the existence of specification errors in the model, while planners search for a short path from the initial state to one of the goal states. Nonetheless, there is rising interest in planners that prove insolvability, and in model checkers to produce minimal counterexamples [224].

In terms of leveraging state space search, over the last decades there has been much cross-fertilization between the fields. For example, based on Satplan [392] *bounded model checkers* exploit SAT representations [50] of the system to be verified, while *directed model checkers* [215, 430] exploit panning heuristics to improve the exploration for falsification; partial-order reduction [628, 296] and symmetry detection [268, 455] limit the number of successor states, while symbolic planners [128, 366, 217] apply functional data structures like BDDs to represent sets of states succinctly.

There are different options for finding optimized schedules with the help of a model checker that have been proposed in the literature. First, in the *Soldier* model of [548], rendezvous communication to an additional synchronized process has been used to increase cost, dependent on the transition chosen, together with a specialized LTL property to limit the total cost for the model checking solver. This approach, however, turned out to be limited for our purpose. An alternative proposal for branch-and-bound search is based on the support of native C-code in Spin (introduced in version 4.0) [547]. One case is the TSP, but the approach is generally applicable to many other optimization problems.

Chapter 25

Further Application Areas

What is AI? In an industrial context you will hear the ones saying *a threat to the world by thinking machines superior to humans*, or other stating *storing a number in the computer means that the machine has learnt.* We think both statements, reflecting either a strong or a weak view on AI, are not pragmatic. In this book we, therefore, align with an efficiency- and performance-oriented interpretation of AI, which we denote as *Algorithmic Intelligence*. The very same AI can beat a human in a competition on a fast machine, and lose on a slower one.

In this chapter we highlight that programming and algorithm engineering remains at the core of the development of intelligence, while aiming at real-world applications. We give insights to the design principles, requirements, and evaluation of machine learning algorithms, and, then, list several potential AI projects for the software industry.

25.1 Introduction

Machine[1] intelligence is increasingly influencing our lives. It has been characterized by Arthur Samuel (1959) as *the field of study that gives computers the ability to learn without being explicitly programmed*. Automated learning, however, is still largely a mystery. During training, we feed AI systems with data, but what conclusions the AI draws from this, according to which logic and which criteria it then makes decisions, is usually inscrutable for people.

A lot of research is still needed to understand how learning models work. And to develop new methods that provide more meaningful contextual information must reflect on how AI systems work, so that people can understand decisions and recognize possible wrong ones and prejudices more easily. Explainability and transparency of AI processes are closely linked and are central goals for governance. Projects should go beyond the previous approaches to meet the requirements of a digitized society. Currently, graphical tools are an important part of this effort, but this solution is nowhere near sufficient. A combination of novel approaches is necessary to understand AI systems. The desire is traceability, explainability and transparency, especially for practical applications. The main aim is to strengthen trust in AI systems, as there is no trust without transparency.

Some methods, such as the shopping basket analysis, offer confidence as a quality measure of rule induction. Confidence speaks of the trust that you put in an inference. For example, if a database is filled with very few case studies, the result of the AI conclusion is often less trustworthy than with a lot of data.

What is the difference of confidence and accuracy?

[1] joint work with Andreas Wulfes, Andree Lüdke, Ashraf Abdo, Björn Schwarze, Hendrik Rothe, Luisa Strelow, Lara Luhrmann, Salome Gindre, Tino Wahler, and Vanessa Just

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3_25

Confidence defines the probability of the event (or probability of input to fall in different classes). If a class has high probability, then it has high confidence. Confidence value can be calculated for single input as well giving the meaning as how much the algorithm is confident for that class. On the other hand, accuracy defines the skill of the learning algorithm to predict accurately. It defines the percentage of correct predictions made from all predictions — Hena Bawa.

Confidence and confusion matrices as evaluation criteria are closely related to correlation and covariances, which measure the interplay of properties. In stochastics, the correlation matrix is a symmetrical matrix that records the correlation between the components of a random vector. The correlation matrix can be obtained from the variance-covariance matrix and vice versa. The correlation matrix can be interpreted as a generalization of the variance of a 1D random vector over several dimensions. It is used, among other things, for the study of eigenvalues and -vectors as in principal component analysis (PCA) and in singular value decomposition, as well as for feature selection and Kalman filtering.

The singular value decomposition (SVD) is at the root of most ways of analyzing datasets that are matrices. One important reason why the SVD is so important is that it is defined for every matrix A with m rows and n columns of rank k. The SVD $A = USV^T$ for (orthonormal) matrices U and T, where U has dimension $(m \times r)$, T has dimension $(n \times r)$. SVD has been extended to matrices with missing entries. The matrix product in the SVD can be simplified to $A = \sum_{k=1}^{r} \sigma_k u_k v_k^T$, where u_k is the k-th column of U, and v_k^T is the transpose of the k th column of V. This simplification decomposes A into the sum of r matrices of rank 1. The SVD is most interesting because of the following property. Let U_k be the k leftmost columns of U, then U_k is an $(m \times k)$ matrix whose columns are the same as the first k columns of U. Similarly, let $V - k$ be the k leftmost columns of V. Let S_k be the $(k \times k)$ diagonal matrix, whose diagonal entries are $S_{1,1}$ to $S_{k,k}$. Define $A_k = U_k S_k V_k^T$ The rank of A_k is k. Eckhart and Young proved in 1936 that $A_k = \min_{X:rank(X)\leq k} ||A - X||_2$, which means that among all matrices of rank k or less, A_k has the smallest squared error with respect to A. The squared error is simply $||A - X||_2 = \sum_{i=1}^{m} \sum_{j=1}^{n} (A_{ij} - X_{ij})^2$ and is sometimes called the Frobenius norm of the matrix $A - X$. In short, the SVD essentially provides the best low-complexity approximation of a given matrix $A \approx U_k S_k V_k^T$.

PCA is a mathematical construction closely related to SVD. It is used in practice to find a more concise representation of the data points during dimensionality reduction. PCA is widely used but has some drawbacks. First, it is sensitive to the scaling of features. Without changing the correlations between features, if some features are multiplied by constants, then the PCA changes. Second, PCA is unsupervised. It does not consider any label that is to be predicted. The directions of maximum variation are usually not the directions that differentiate best between examples with different labels. Third, PCA is linear. If the data points have a nonlinear structure, PCA will not capture it.

To assess how well a trained model fits the data, in practice one looks at the following two metrics:

Sensitivity: the likelihood that the model will predict a positive outcome for an observation when the outcome is indeed positive.

Specificity: the probability that the model will predict a negative outcome for an observation when the outcome is indeed negative.

Terms often used are positive (P): number of real positive cases in data; negative (N): number of real negative cases in data; true positive (TP) = hit; true negative (TN) = correct rejection; false positive (FP) = false alarm, type I error or underestimation; and false negative (FN) = type II error or overestimation.

Other values are derived: recall, hit rate, or true positive rate (TPR); selectivity or true negative rate (TNR); precision or positive predictive value (PPV); negative predictive value (NPV); miss rate or false negative rate (FNR); fall-out or false positive rate (FPR); false discovery rate (FDR); false omission rate (FOR); accuracy (ACC); balanced accuracy (BA); and F1 score.

In the first step, we start with monitored learning, in which we have training examples that have been preclassified by experts and whose occurrence in new, yet unknown data sets or a hazard value can be identified.

For validation, the data is divided into test and training set. A simple, first supervised learning process can be generated from a metric distance measure and a set of so-called labeled data. In case-based reasoning such as

k-nearest neighbor, a reconciliation of existing labeled data for a new data point is suggested. Many classifiers are linear, i.e., they are based on the separation of the positive and negative examples in a vector space with a linear parting plane.

With the so-called *kernel trick*, simple linear separation methods can be applied to data that cannot be linearly separated. Here, the feature vector in the learning process, be it the simple linear classifier or the support vector machine, the perceptron, or the alike, is lifted into a higher-dimensional space, where this separation is possible.

An easy way to visualize these two metrics is to create a *receiver operating characteristic curve* (ROC curve). This is a graph that shows the sensitivity and specificity of a logistic regression model. A model with high sensitivity and high specificity has an ROC curve that is oriented towards the upper left corner. A low sensitivity, low specificity model has a curve that is close to the 45-degree diagonal. The *area under curve* (AUC) gives us an idea of how well the model can differentiate between positive and negative results. The AUC can be between 0 and 1. The larger the AUC, the better the model can classify the results correctly. We will be using AUC to compare the performance of two or more models. The model with the higher AUC is the model with the best performance.

In unsupervised learning, there is no training data with predefined classifications that can be used for orientation. These can be found, for example, through clustering. In addition to the feature vector, a distance measure, a so-called *metric*, always plays a role. Simple metrics for numerical data include the Euclidean distance, derived from the so-called 2-norm, or another p-norm with $p \neq 2$ (e.g., *absolute-sum* or *infinity*-norm). In the case of strings, the Levenshtein or edit distance could help, which turns out to be the set of editing operations and can be found with dynamic programming.

Of course, linguistic peculiarities can also be used here (removal of fill words), etc. A frequency analysis in texts (bag of words) can also lead to a measure of distance (often used with spam filters). For time series there are similar distance measures as those of the *dynamic time warp* (DTW) procedure, which is based on the approximate character string search. There are similarities between DTW and the Needleman-Wunsch algorithm (global alignment) and the Smith-Waterman algorithm (local alignment). Distances between more complex data, e.g., in tree structures or graphs, can also be found using feature vectors. As soon a metric space (M,d) with real-valued distance measure d is available, machine learning processes can be used.

Mathematically, a mapping $d : M \times M \to \mathbb{R}$ is called a *metric* on M if the following conditions are satisfied for any elements x, y and z of M: (1) positive definiteness: $d(x,y) \geq 0$ (2) symmetry: $d(x,y) = d(y,x)$ (3) triangle inequality: $d(x,y) \leq d(x,z) + d(z,y)$. This applies to all distance measures considered above; also note (1) follows from (2) and (3). E.g., for clustering, inter- and intra-cluster distances are measured and linked to a target value that needs to be optimized for an at least locally optimal clustering (k-means, k-medoid). Also hierarchically organized agglomerative clustering (e.g., with thresholding dendrograms); or density-based clustering methods are based on a suitably selected distance measure. Whether the metric refers to a string problem or a tree distance problem, a time series or a combination of several submetrics, is dependent on the internal and extended, enriched representation of the problem at hand in a suitable data structure.

Regularization is the method for avoiding model parameters growing into the extreme, they are usually included into the loss function for the error to be minimized. Consider, a matrix factorization with model parameters for each person and for each movie, and a training set of 500,000 ratings for 10,000 viewers and 1,000 movies. A rank-50 unregularized factor model is almost certain to overfit, as the number of parameters is larger than the number of data points for training. We can typically improve generalization by using a large value for the rank, while reducing the effective number of parameters via regularization.

There are numerous application areas. Our selection indicates a mixture of algorithmic solutions to project problems, and gives some hindsights on what further questions might arise in daily practice within the body of a larger software company.

The actual meal may look different on every plate but the core of intelligent inference in practice remains algorithmic. In our daily project work in the software industry, we crossed the following topics. They can serve as on an outlook, on what the work on the algorithmic foundation of artificial intelligence is capable of, and where it may head to. As a matter of being revenue drivers for real companies, this time we do not go into full

algorithmic details. Even if not shown to the reader, for each of the topics, however, there is a working POC implementation (mainly as Jupyter Notebooks in Python using ML libraries).

Before we dive into describing the problems, let us meditate. What are additional requirements for an AI in practice?

Freedom from discrimination: AI systems should treat everyone in a non-discriminatory and fair manner. For example, if AI is used as a guide for medical treatment, loan making, or job dying, then the same recommendations apply to all people with the same symptoms, conditions, or professional qualifications. AI developers and be made aware of how (human) prejudices find their way into AI. And they should themselves reflect the diversity of the world in which we live.

Reliability: AI systems must be able to operate reliably, securely and continuously, and not only under normal conditions, but also under unexpected circumstances or when they are attacked. Rigorous testing during development and implementation is critical. People should always play a pivotal role in making decisions about how and when to use AI systems.

Protection of privacy: Like other cloud technologies, AI systems also comply with applicable data protection laws. These require transparency in the collection, processing and storage of data and provide suitable control mechanisms for consumers so that they can decide for themselves how their data is used. AI systems may only use private data in accordance with applicable data protection standards and must protect data securely against theft.

Accessibility: For AI to benefit everyone, it must also incorporate human needs and experiences. AI systems should generally be designed to understand the context, needs, and expectations of the people who will use them. Design principles geared towards accessibility and inclusion help developers identify potential barriers.

Transparency: AI will affect people's lives, so we should say when, and provide contextual information about how, AI systems work, so that people can understand AI decisions and recognize possible wrong decisions and prejudices more easily.

Accountability: As with other technologies, with AI systems, the people who die, who develop and deploy them, are responsible for their use. When creating accountability standards for AI, we should draw on expertise and best practices from other areas such as healthcare and data protection.

We see that the demands on an AI system are considerably high. In our opinion, full transparency is too high a requirement. Self-explanatory AI systems are very difficult to design. How can a deep neural network that has trained thousands and thousands of weight vectors and represents a complex graph-based real-value function as a black box, explain why it recognizes what and how? Nonetheless there are neuro-symbolic approaches that help to reason about data-driven inferences via adding symbolic representation and reasoning of knowledge.

After this brief reflection, we are ready to go on. In the following we give some insights to use cases and potential projects.

25.2 Vacancy Simulation and Temporal Pattern Mining in Smart Homes

Home automation is building automation for a home, called a smart home or smart house. A home automation system will monitor and/or control home attributes such as lighting, climate, entertainment systems, and appliances. It may also include home security such as access control and alarm systems. When connected with the Internet, home devices are an important constituent of the Internet of Things (IoT).

There is a growing application for machine learning based on data that is logged in relational DBs of smart home router devices. This data can be extracted and exported and can be used to generate patterns for automated control when leaving the house, or to detect temporal rules of frequent behavior. There are algorithms like

PivotMiner for short patterns and *CTD* for complex disjunctive temporal patterns in form of a tree. Some inferences are costly and have to be tuned to yield an acceptable speed. The crucial measure for these algorithms is the confidence in the application of rules. It is also possible to compare the computer-generated rules with the ones of the users.

25.3 Form Filling

Filling online forms can be a tedious task, as this often is repetitive and time-consuming. When OCR scans of orders are provided and keywords are detected, there is still the problem of *missing values*. The AI must predict these, based on historical data. This *data imputation problem* is a machine learning task to predict the values in each cell. It is usually solved via an iterative (boosting) approach, beginning with simple values like the mean of existing data and then using machine learning with the output values being the entry of a requested cell. The classification values are judged either by given data or by user feedback, which in turn lets the algorithm learn new values. Additional attention has to be given to the fact that some data like addresses becomes outdated.

25.4 Demand and Sales Prediction for Retail Trade

Financial forecasting is essential for all companies, especially in retail trade. The input are large databases of weekly/monthly turnover sales, e.g., partitioned in many individual shops of a bigger retail company, and the predictions wanted are the sales of the next month, allowing to also predict the upcoming demand. The automatically inferred model can be compared with the model that is manually drawn by sales experts in the company, which also allows to select and reduce the essential features for the machine learning process. Tools like singular value decomposition and principal component analysis apply. Time series prediction algorithms approaches like various moving averages are applicable.

25.5 Geo-Location Tagging for Better Goods Assortment

No doubt that the surrounding infrastructure in terms of schools, hospitals, shops, highly populated vs urban area etc. have a large impact on the shopping behavior of people especially in certain stores (e.g., fashion, daily needs). To combine previously web-crawled and subsequently tagged geo-location data together with the goods assortment is of high importance for the distributor/seller. To interplay of spatial knowledge with the prediction of the demands merged into one algorithm requires a combined machine learning classification algorithm and several statistical stages of clustering as well as the application of correlation and covariance matrices. Afterwards, the results can be visualized in an annotated (open street) map.

25.6 Automated Inventory List Creation with Vision and Speech

Compiling inventory lists is a tedious task especially for the combination of searching, displaying and note-keeping. Suppose that we have a big wardrobe, with employees that must take pictures of all clothes and classify, which type, color, size, texture and further features the fashion has, e.g., for which person age and sex a fashion object suits. This process can be half-automated by using neural nets for vision, and further algorithms for

speech detection. While there are some precompiled networks for vision and speech, they must be adapted to meet this task.

25.7 Temporal Task-Motion Planning for Robots with Resources

There is an emergent need to enhance the capabilities of robots deployed in factories, warehouses, and other logistics operations. In these settings, robots are required to pick up and deliver objects from multiple locations within certain time frames. This gives rise to challenging task-and-motion planning problems, as it intertwines logical and temporal constraints about the timing of operations with geometric and differential constraints related to obstacle avoidance and robot dynamics. Due to the limited load capacity, a robot may have to deliver some objects before being able to pick up others. As a result, the robot may have to travel longer, which could violate the temporal constraints. When not all tasks can be completed within the specified time frame, the robot also must decide which tasks to abandon, while seeking to complete as many of the remaining tasks as possible. Autonomous robots also need a reliable way to preserve their energy level and to recharge, while performing a task such as inspection or surveillance. Toward this objective, we consider the multi-goal motion-planning problem with multiple recharging stations, where a robot operating in a complex environment must reach each goal while reducing the travel distance and the number of times it recharges. We search over a discrete abstraction, which is obtained via a probabilistic roadmap, and use a reward function to calculate when, where, and whether it is beneficial to recharge. This results in short tours that also reduce the number of recharges. Such tours are used to guide sampling-based motion planning as it expands a tree of collision-free and dynamically feasible motions. For nonlinear dynamical robot models operating in obstacle-rich environments the efficiency of the approach is highlighted.

25.8 Optimizing Integrated Production and Route-Planning

Given several production units located on some map, the task might be to plan the combined production and delivery of goods according to the dynamically incoming orders including the time for the delivery. Such production units often run 24/7 at different locations and have certain capacity limitations. The production might be split into several chunks and should be small. Even the production waste might be sold and must be accounted for. The distance to the customer has an impact on the cost, so that route planning must be linked to the selection of the production venue. There are two options to solve the problem: one is Monte-Carlo tree search (e.g., with NRPA), the other is rather classical Operations Research (OR) via Mixed-Integer Linear Programs (MILP), with a two-stage approach. In the first stage a possible fragmented solution must be found that respects the ordering and (per day) resource constraints, finding an assignment from units to he orders, the second stage optimizes order execution in the units for minimizing their split.

25.9 Safe Travel Recommendation with Movement Data

In the touristic industry there are many data-driven inference task. If bookings are centrally stored, then refined algorithmic solutions are needed to predict the change of customer's booking behavior based on historical data and incoming news available in different multimedia sources. Post-COVID there still will be the need to track the flow of people e.g., in a resort to balance load and to prevent infections. Tracking data can be used for mapping. Algorithms can infer if traces of people have an overlap in time and space. Interestingly, the *color nearest–neighbor problem* solves this problem in sorting time $O(n \lg n)$; the number of colors (in the set of

traces) does not matter. This is also applicable to determine, if some salesmen could have met during their travel.

25.10 Improved Static Code Analysis with Machine Learning

In the areas of formal methods and software engineering, there are several algorithms to perform static analysis of code. This is used to detect redundant code or programming rule violations. This is usually done following variable dependencies to remove possible flaws. e.g., in device drivers. One newer but highly demanding request is to find security issues in the code, i.e., which contain vulnerabilities for intrusions like buffer overflows. It is thought as an addition for the compiler that warns the programmer of producing code that may enable malicious attacks. Machine learning algorithms based on textural pattern detection and graph structures, like abstract syntax tree, call graphs etc. apply. The detection does not have to be perfect but to provide a low false positive rate and a timely response during coding. Engineered algorithms are needed, also because of the heterogenous data structures (graph, trees, strings) to be learnt. To apply machine learning algorithms, graph embeddings and traversals are key ingredients.

25.11 Contract Prediction on Account Transaction Series

Another interesting task is to infer existing contracts of a customer on the mere basis of account transactions data. The data might be present at bank institutions and been provided by the customer while using account apps. This can be exploited for the advertisement of alternative contracts. While mostly monthly and simple, transactions could also be weekly or irregular. Machine learning can be used to predict the next transaction that will be executed, and a series of transactions may be clustered to a contract (e.g., insurances, rent, electricity bills, regular payments) as one means for offering people better deals. The algorithmic challenge is to include background knowledge about the kind of vendor, their interconnection; sometimes even the inclusion of geo-localized data helps. As transactions are often done electronically, one could even infer more complex financial patterns in the user behavior.

25.12 Recommender Systems for eCommerce

Recommender Systems are undoubtedly one of the major revenue drivers in online shops and eCommerce. Besides WalMart, who was first to analyzed shopping bills, it was Netflix that broke through with recommendation of films to customers. Nowadays, based on such collaborative filtering algorithms at Amazons and other online shops, there always is recommendation based on analyzing link, purchase, and GUI interaction. Both have application to recommend articles for sale, e.g., in an online fan shop. On top of users given only a session ID, another problem to solve is the generation of stem product numbers for better generalization of rules. Another problem is that the bills in smaller online shoppings are often sparse, so that some more preprocessing must be done to apply association rule mining algorithms. Furthermore, dynamic information must be used for *dynamic pricing*.

25.13 Predictive Maintenance to Produce Assemblies

The problem in this project was to predict upcoming maintenance needed within the steel production to prevent an early the wearout of motors of the welding units. A vector of sensory information on the production process and the actuators is continuously recorded, and historical data on maintenances is given in separate files. The inference task boils down to do time series prediction. One additional obstacle is that there often is a time shift in between the time series patterns. There are several different options, one is the generation of sliding windows for the time series to allow traditional learning algorithms to be executed, another is case-based reasoning based a distance metric for time series *dynamic time warp*. Side issues are detecting and removing trends and finding periods of repetition automatically via a Fourier transformation.

25.14 Knowledge for Retail

As a unique opportunity to support smart retail use cases may include intelligent intra-logistics, branch setup optimization, retailing with smart refrigerators, and service robots. All aspects come with the goal of optimizing and automating the merchandise management, as well as to provide the customer with detailed advice from employees and a new shopping experience. A basic technology are so-called digital semantic twins, which digitally mirror the stationary branch. This enables robots, for example, to fill shelves automatically, or to set up individual stationary markets that adapt to customer behavior.

25.15 Industrial Inspection

In many areas of industrial engineering parts of high-end assemblies were inspected with microscopes. In a resolution of micrometers, the images of produced parts are examined and stitched, to discover defects or obscuring particles. For this to work effectively, companies apply automated imaging. To reduce the working load of the human microscope operator, an individual inspection plan is generated for every object shape. The task is to find an algorithm to minimize the number of high-resolution images for inspection for objects of various shape, and to generate a list of inspection waypoint, for which a compete cover is generated. Minimizing the number of *shots* results in saving time and resources during object surface quality control in high-end manufacturing. Similarly, high-resolution pictures arise from stitched images of drone images during the automated flight inspection of the blades of windmills, of bridges, or buildings, etc. The task is to find defects in the surfaces fully automatically, or at least to lower the number of images for manual validation. If the motion planning problem is resolved for the drone, different classification algorithms from computer vision are needed.

25.16 Skill Recommendation

One last possible application in our small selection is to develop a recommender system for skill suggestions in employee training. The general recommendation would be of the type *You have skill X and Y. Employees with these skills are likely to also have skill Z.* Such recommendation can be used for staff training, or in managing project teams and tribes. Such skill recommender could cover: a) the identification of significant patterns between employee skills and b) the generation of recommendations using suitable algorithms. The research question would be: *how can a recommender system based on collaborative filtering generate skill suggestions for continuous employee training?* Different matrix factorization methods apply, but also *page rank approaches* can be used.

25.17 Bibliographic Notes

There are several books that dwell on machine learning, a basic selection is

- Stuart Russell and Peter Norvig. *Artificial Intelligence – A Modern Approach.* Prentience Hall, 2nd Edition (2003).
- Peter Flach. *Algorithms that Make Sense of Data.* Cambridge, 1st Edition (2014)
- Tom Mitchell: *Machine Learning.* McGraw Hill, (1997) + New Chapters (Web).
- Pat Langley: *Elements of Machine Learning.* Morgan Kaufmann (1995).
- Shai Shalev-Shwartz, Shai Ben-David. *Understanding Machine Learning.* Cambridge (2014).
- Christopher M. Bishop: *Pattern Recognition and Machine Learning.* Information Science and Statistics, Springer (2006).
- Richard Sutton, Andrew Barto: *Reinforcement Learning,* MIT. (1998)
- Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, Vipin Kumar: *Introduction to Data Mining*. Pearson, 2nd Edition (2019).

Excellent online resources include Andrew Ng's *Stanford Machine Learning* lectures. We also highlight the work by Charles Elkan on predictive analytics and data mining. Moreover, there is a continuously and rapidly growing body of publications at AI conferences (e.g., AAAI, IJCAI, ECML, ICML, KR, NIPS, SOCS, ICAPS). Of course, journals like the *Artificial Intelligence Journal* or the *Journal of Artificial Intelligence Research* also publish exciting research.

One IBM study revealed that 82 percent of companies would like to use *Artificial Intelligence* to increase sales. However, 60 percent of the total of 5,000 surveyed decision-makers stated that they were concerned that they would be held responsible for possible wrong decisions by the systems.

The criteria for a good learning process are often measured by the accuracy of their prediction. Alternatively, criteria such as precision or yield are combined. We are often familiar with this problem of different negatives from statistically ambiguous relationships in Corona tests.

Software implementations of classification, clustering and recommendation algorithms include R, Weka, the ScipY and Scikit-Learn, Implicit and Surprise Python libraries, Orange, MATLAB,SPSS,SAS, STATA, SQL Server Analysis Services, and many, many more. The reference for the PivotMiner for the inference on sequential short patterns is [325], while the one CTD for complex disjunctive temporal patterns is [498]. The colored nearest neighbor solution can be found here [4]. There are many more. We suggest the reader to do some *literature snowballing* on his/her own.

Index and References

Index

ε-NE, 327
ε-Nash equilibrium, 327
∞-norm, 431
k-ary heap, 38
k-means clustering, 431
k-medoid clustering, 431
k-nearest neighbor, 430
k-nearest neighbors algorithm, 316
kd-tree, 153, 164, 182
n-gram, 104
p-norm, 431
0/1 transformation, 172
10-fold crossvalidation, 125
15-Puzzle, 19
2-OPT, 397
2-SAT, 117
2-player normal-form game, 326
2D packing, 361
2D rectangle packing, 373
3-OPT, 397
3D, 45
3D packing, 361
3D packing problem, 362
3D packing with object orientation, xvi
3D printing, xiv, 377
3DP, 377
50% Chess, 256

a priori time limits, 267
A*, 35, 46, 391
A* with consistent heuristic, 392
AABB, 376
ABD, 135
abduction, xv, 257, 258, 271, 279
abductive inference, 258, 271
abductive planning, 278
aberrant behavior detection, 135
ablation-type study, 270
ABS, 411
absolut-sum norm, 431
abstract state space, 207
abstract syntax tree, 435
abstraction, 258
abstraction heuristic, 260
acceleration, 401
accessibility, 432
accountability, 432
accumulated relevance, 125
accuracy, 119, 125, 161, 429, 437
Ackermann's function, 7
action planning, viii, xiv, xv, 257, 427
activity, 412
Adam optimizer, 77
adaptive boosting, 128
adaptive sampling, 379
adaptive stationary, 436
additive boosting, 128
additive manufacturing, 377
addressable priority queue, 39
adjacency list, 36
adjacency matrix, 36
admissible heuristic, 401, 420
adversarial agent, 325
adversarial domain, xvi
adversarial environment, 325
adversarial planning, xvi
adversary mixed strategy, 330
affine gap cost, 350, 359
affinity propagation, 312
agent response, 325
agent-based simulation, 411
agglomerative clustering, 431
aggregate-combine GNN, 185
aggregation function, 185
Aho-Corasick algorithm, 132
AI planning, 193
AI search, 46
algebraic decision diagram, 257
ALGOL, 33
algorithm engineering, viii, 33, 386
algorithmic intelligence, vii, xi, xiii, 411
all approximately nearest neighbor problem, 155
almost zero-sum game, 326, 330
AlphaGo, 68, 84
ALS, 188
alternating least square optimization, 181
alternating least squares, 188
altitude, 167
Amazon, xiii
American Checkers, 281
AND/OR search tree, 225
angular change, 394, 397
anomaly, 132
anomaly detection, 150
anomaly immune system, 151
ANSI C, 33
ant algorithm, 104
antivirus program, 129
any-angle packing, 375
anytime algorithm, 373
AP, 24, 361, 365, 393, 402
aperture problem, 153
approximate best response, 335
approximate bisimulation, 260
approximate knowledge-based paranoia search, 228
approximate medial axis, 378
approximate paranoia search, 228
approximate string matching problem, 305
approximately k-nearest neighbor problem, 155
approximation algorithm, 393, 409
Apriori, 315
area under curve, 431
art gallery problem, 388
Art of Computer Programming, 33
artificial immune system, xii
artificial intelligence, viii, xii
Artificial Intelligence Journal, 437
assembly, 418
assembly-line network, 414
assignment problem, 24, 361, 365, 393, 402
association rule, 191, 309, 315, 435
assumption-based truth maintenance system, 280
asymmetric game, 199
asymmetric TSP, 361, 391
asynchronous progress, 414
ATMS, 278
atomic transition, 420
ATSP, 361, 391
attention, 74
attribute, 412
AUC, 431
audio-visual files, 119
audio-visual material, 119
auto correlation, 134
automated event, 120
automated intelligence, 429
automated planning, xvi, 325
automated selection of abstraction, 278
autonomous decision making, 298
autonomous logisitics, 308
average scoring, 237
average Seeger score, 241
Awari, 194, 208, 215
axis-aligned bounding box, 376

Backgammon, 86, 104, 256
background knowledge, 435
backpropagation, 67, 183
backtracking, 13, 221, 225, 226
Backus-Naur, 33
backward breadth-first search, 272
backward chaining, 272
backward inference, 273
backward search, 259
backward uniform-cost abduction, 272
bag of words, 431

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3

balanced accuracy, 430
BAliBASE, 356
ball vector machine, 164
bandid learning, 191
bandit-based search, viii, 103
bandwidth of Gaussian, 159
bandwidth of kernel, 159
base map generation, 165
basket analysis, 429
Bayes classifier, 161
Bayes network inference, ix
Bayes' network, 191
BDD, 193, 194, 209, 257, 284
BDD compression ratio, 271
BDD hashing, 213
BDD inference, 272
BDD perfect hashing, 216
BDDA*, 257, 261, 263
beam, 86
Beam MCS, 86
Beam NRPA, 86, 92
behavioral state, 298
belief set, 225
belief state, 290
belief state space, 281
belief state tree, 290
Bellman-Ford algorithm, 409
Bellmann and Ford's algorithm, 392
Belote, 251
best matching feature, 142
best response planning task, 330
best-first locality, 208
best-first search, 347
best-fist ELO algorithm, 255
best-response equivalent, 327
best-response set, 327
BGRU, 185
biased regularized incremental simultaneous matrix factorization, 188
bidding, 217
bidding stage, 218
bidirectional BFS, 262
bidirectional search, 259
bidirectional sequence model, 185
big data, 107
big data analysis, xv
bijection, 22
bin packing problem, 362
bin-packing algorithm, 268
binarization, 154, 157, 163
binary classification, 119
binary decision diagram, 193, 194, 209, 257
binary heap, 35, 37, 38, 50
binary kernel, 158
binary search, 27, 225, 233
Binomial coefficient, 9
birth-giver agent, 300
bisimulation, 260
bisimulation reduction, 278
bitmap image, 398
bitshifting, 222
bitstate hashing, 337, 340, 348, 425
bitstate-based duplicate detection, 347
bitvector machine, 153, 157, 312
bitvector set representation, 232
black box, 432
block transfer, 341
BlockQuicksort, 58, 66
Blocksworld, 194
Bloom filter, 340, 348
BLSTM, 185
BnB, 365, 420
board, 72
Boltzman machine, 186
Boltzman softmax, 397
Bolzman Machine, 191
booking behavior prediction, 434
Boolean formula, 110
boosting machine learning, 433
border, 54
border blocking, 127
bottom heap, 54
Bottom-Up-Heapsort, 50
bounded model checking, 428
bounding, 420
box packing, 363
branch misprediction, xiv
branch-and-bound, 225, 420
branch-and-bound optimization, 420
branch-and-bound search, xv, 399
branch-and-bound technique, 409
branching, 420
breadth-first explicit-state model checking, 340
breadth-first search, 262, 340
breadth-first state-space generation on the GPU, 348
Bridge, 217, 251
BRISMF, 188
BRNN, 185
broadcast, 167
broadcasting industry, 127
brute-force backward search, 268
bucket elimination, ix
bucket heap, 41
bucket-based priority queue, 39
buffer overflow, 435
building growing, 376
building standing cards, 233
burned pancake problem, 198
business intelligence, vii
BVM, 153
by-hand, 231, 232

C, 3
cache, 167, 318
cache efficiency, 45
cache hierarchy, 43
caching policy, 169
CAD model, 383
call graph, 435
candidate matrix, 184
canonical pattern combination method, 268
canonical pattern database, 269
CAPG, 325
carbon footprint, 297
card group, 230
case-based reasoning, xv, 309, 315, 316, 322
Catch a Mouse, 289
categorization, 131
Cayley graph, 215
CB, 310
central processing unit, 185, 194
CF, 310
chaos theory, xii
checker-board arrangement, 159
Checkers, 194, 213, 284
checking enabledness, 340
Chess, 194, 213, 284
child-sibling representation, 38
Chinese Checkers, 215
Chomp, 289
Christofides' algorithm, 393
chromatic number, 107, 114
classical planning, 334
classification, 316
classification algorithm, 433
clause learning, 117
client-server architecture, 166, 298
Clobber, 289, 296
Closed list, 195, 259, 343
closed list, 207
closed-world assumption, 275
closing situated nodes, 175
cluster computation, 347
clustered PTSP, 390
clustered traveling salesman problem, 387
Clustered TSP, 399
clustering, 165
clustering algorithm, 433
CNF, 110
CNN, 67
coalesced memory access, 339
coarsest bisimulation, 260
CoD, 310
coherence matrix, 280
Coil-in-the-Box, 100
Coin Flipping, 283
collaborative, 181
collaborative filtering, viii, xv, 182, 187, 188, 191, 310, 322
collaborative map generation, 165, 177
collision counting, 382
collision-free and dynamically-feasible motion, 434
collision-free packing, 377
color nearest-neighbor problem, 434
colored tour, 401
coloring, 108
combinatorial auctions, ix
combinatorial game, xiv
combinatorial game theory, 230, 254, 255
combinatorial optimization, 46
combinatorial problem, xiv
combinatorial search, viii
combine function, 185
combined task and motion planning, 401
combined vision and speech recognition, 433
comma-separated value file, 69
communicating processes, 417
communication arc, 299
communication parameter, 299
compare-and-swap, 425
competing agent, 325
complementary pattern databases, 268
complementary PDB creation, 278
Complementary Planner, 267
complete heap, 52
complex behavior, 298
complex TSP, 104
computational biology, 359
computational intelligence, viii, xii, xiii
Compute Unified Device Architecture, 348
computer cache, 43
computer game playing, 406
computer Go, 230
computer vision, viii, xiv, 153
computer word, 24
Concentration, 294
concept learning, 191
Conditional Random Field, 129, 130
conditional random field, viii, xv, 150
confidence, 315, 429, 432
configuration, xv, 314
configuration problem, 314
configuration task, 309
conflicting goals, 325
confusion matrix, 430
congestion game, 334
conjunctive normal form, 110
Connect Four, 193, 209, 216
connected component, 174
consistent heuristic, 78, 392
constant color frame, 125
constraint, 137
constraint propagation, 110, 309
constraint satisfaction, viii
constraint satisfaction problem, xv, 314, 373
constraint TSP, 104
container loading, 376
Container Packing, 104
container packing, xiv, 361, 363, 375
content-based filtering, 310
content-based recommendation, 322
contextual information, 429
contract limit, 225

contract prediction, 435
controller, 390
conversion, 209
convolution, 68, 74
convolutional feed-forward neural network, 67
convolutional neural network, 67
cooperative game, 217
cooperative pathfinding, 104
coordinated conservative synchronization, 308
Corona, 437
correct algorithm, 227
correctness verification, 428
correlation, 433
correlation engine, 131
correlation matrix, 123, 430
Cosine distance, 311
cost optimization, 420
Cost-adversarial planning game, 325
cost-based planning, 258
cost-optimal abduction, 257
cost-to-goal, 83
counterexample, 420
counterfactual regret minimization, 255, 256
counterplanning, 335
covariance, 433
covariance matrix, 123, 188, 430
coverage, 80
CPU, 185, 193, 201
creation phase, 168
critical action, 333
cross-validation, 160
Crossword Puzzle, 90
CSP, 314
CSV file, 69
CTSP, 365, 388, 399
CUDA, 68, 186, 348
current temporal relation, 145
curriculum learning, 67, 76
curse of dimensionality, xv, 154, 159, 164, 310
cybersecurity, 325

DALL-E2, 164
data imbalance, 145
data imputation, 433
data mining, 119, 128, 322, 437
data stream, 124
data-driven AI, 432
database index, 49
Davis-Putnam-Loveland-Longmann, 110
DBSCAN, 312
deadline for action, 333
deadlock, 422
deal, 217
decision diagram, 261
decision list, 316
decision support system, 119
decision-making, viii, xvi
declarer, 218
decrease-key, 37
deep learning, 67, 127, 163
deep neural network, xiv, 83
Deep RRT*, 389
DeepMind, xiii, 84
DeepStack, 296
degradation of GPS signals, 168
degree of parallelism, 125
Delaunay hierarchy, 153
Delaunay triangulation, 153, 164
delay, 412
delayed duplicate detection, 213, 343, 347
deliberative reasoning, 325
delivery and backhaul, 362
delivery and pick-up, 362
demographic filtering, 310
demographic recommendation, 322
Dempster-Shafer theory, xii
dendrograms, 431
depth-first belief state search, 291
depth-first branch-and-bound, 361, 365, 388, 401, 409, 420
depth-first search, 401, 420
DES, 411
description logic, 137
descriptor, 163
DESS, 411
determinant, 397
deterministic algorithm, 26
deterministic step, 420
deterministic time, 412
DFBnB, 388, 401
DFBSS, 291
DHL, 362
diagnosis, 280
diagnosis problem, 280
differential constraint, 434
digibeta dropout, 127
Digital Betacam, 125
digital map generation, 165
digital semantic twin, 436
digital twin, 436
Dijkstra's algorithm, 274, 391, 392
Dijkstra's method, 35
dilatation, 170, 171
DIMACS, 118
dimensionality reduction, 311, 430
diminishing return, 268
directed model checking, 278, 347, 428
Dirichlet process mixture model, 311
discrete event system, 411, 425
discrete event system simulation, 411
discrete-time Markov chain, 339
discriminating color, 170
disk-based search, 213
dispatched operation, 340
distance, 317
distance correlation, 311
distance measure, 316, 431
distance-preserving abstraction, 260
distributed intrusion detection system, 150
distribution of cards, 233
divide-and-conquer, 5
DL, 67
DNA, 349
DNA alphabet, 349
DO algorithm, 325
domain, 120
domain theory, 271
domain-independent algorithm, 107
domain-specific problem, 107
dominated solution, 369
Dots-and-Boxes, 194, 208
double-dummy miniglassbox solver, 232
double-dummy Skat solver, 254
double-dummy solver, 217
double-oracle algorithm, 325, 327, 335
Douglas-Peuker algorithm, 175
downsampling, 68
DPLL, 110
dropout, 125
dropping gain, 220
DSKV, 238
DTMC, 339
Dual-Pivot Quicksort, 57, 66
duplicate detection, 286, 340
duplicate elimination, 340
dynamic Bayesian network, 279
dynamic packing, 375
dynamic pricing, 435
dynamic programming, 6, 132, 350
dynamic standing card, 231
dynamic time warp, 132, 431, 436
dynamically feasible trajectory, 387

EBM, 377
edit distance, 359, 431
edit distance problem, 132, 305
efficient perfect hash function, 195
electron beam melting, 377
ELO development, 240
ELO ranking system, xv, 218
ELO ranking system for two players, 239
ELO rating, 237
ELO rating system, 236
empty sequence, 349
end fingerprint, 125
endgame, 217
endgame card, 224
endgame solver, 230
energy-award routing, 434
ensemble of experts, viii
enterprise security manager, 130
entity, 137, 412
Entscheidungsproblem, 33
environment, 402
environmental agent, 298
equivalent card pruning, 226
erosion, 170
erosion operation, 170
error function, 431
error rate, 125
error, trend, seasonal, 134
ESM, 130
ETS, 134
Euclid's algorithm, 9
Euclidean distance, 311, 431
Euclidean metric space, 397
Euclidean space, 391
evaluation, 350
evaluation function, 71
evaluation weight, 122
event, 120, 412
event confidence, 120
event correlation, 120
event evaluation, 120
event learning task, 121
event length, 120
event matrix, 123, 125, 126
event matrix score, 123
event queue, 411
event type, 120
event-driven multiagent-simulation system, 308
evolunationary computing, xii
evolutionary computing, viii
evolutionary learning, 191
exact computation, 381
exact medial axis, 378
expert card, 224
expert game, 217
expert knowlege, 130
explainability, 429
explanation, 258
explanation layer, 143
explicit-state model checking, 337, 347
explicit-state model checking on the GPU, 348
explicit-state retrograde analysis, 212
exploding gradient problem, 183
exploitation, vii, 86
exploration, vii, 86
exponential smoothing, 132, 133
express bus, 341
extended Seeger system, 238
externalization, 41
extract-min, 37
extractability, 386
extreme point, 368
extreme programming, 33

F1 measure, 125
F1 score, 430
Facebook, xiii
factor search, xv
factorial base, 196

factorial number, 6
factorized card-game solving, 253
factorized heap, 41
factorized priority queue, 36
factorizing the search, 230
fall-out, 430
false omission/discovery rate, 430
false positive rate, 435
fault diagnosis, 275
feature, 182
feature extraction, 119, 120, 126, 128, 254
feature scaling, 430
feature vector, 153, 188, 431
FFD, 268
FFI, 268
Fibonacci heap, 35, 37, 38
Fibonacci number, 6
FIFO channel, 417
FIFO logic, 168
file-crossing event, 127
final decision making, 142
financial forecasting, 433
finantial pattern, 435
find-min, 37
fine heap, 51
fingerprint, xv, 122
fininte-domain variable group, 259
finite state machine inference, ix
firewall, 129
first-fit decreasing, 268
first-fit increasing, 268
first-order logic inference, 280
fitness, 398
fitness selection, 91
fixed card, 225
fixpoint operator, 272
flexible GUI, 169
FLM, 377
flood fill algorithm, 36
flood-fill, 43, 46
flood-filling, 43
flow heuristic, 278
flow manufacturing system, 414
flow shop scheduling, 427
flowchart, 188
FOND planning, 334
Fore and Aft, 199
forehand, 225
form filling, 433
formal language, 182
forward chaining, 272
forward inference, 273
forward model, 85
forward search, 259
Fourier transform, 436
Fox-and-Geese, 193, 194, 199, 215
FP-Growth, 315
FP-tree, 315
fractal, 154
fractal computing, 153
Fractals, 4
fractional cascading, 376
fragment elimination, 170, 171
fragmented memory allocations, 374
freedom from discrimination, 432
frequency analysis, 431
frequent pattern, 128, 315
frequent pattern tree, 315
Frobeneous norm, 430
Frogs-and-Toads, 193, 194, 199
front-end, 166
frontier search, 207
FSM behavior, 299
FSM learning, 191
full Delaunay hierarchy, 164
fused layer modeling, 377
fusing approach, 138
future event list, 412, 422
fuzzy control, viii
fuzzy ogic system, xii
fuzzy pattern matching, 138

game, 391
game description language, xv, 281
game encoding, 110
game factor, xv, 217
game graph, 45
game of chance, 218
game of skill, 218
game opening, 217
game playing, viii
game-theoretic method, 334
game-theoretical value, 68, 194, 221, 226
gap, 349
gap closing, 170, 171
gap extension, 350
gap opening, 350
gap-only column, 353
gate, 184
gate operation, 184
gated recurrent unit network, 184
Gaussian algorithm, 11
Gaussian elimination algorithm, 10
Gaussian kernel, 157
GBFS, 83
GCD, 9
GDL, 281
GDL semantics, 282
GDL syntax, 282
GDL with incomplete information, 281
GDL-II, 281, 283
general game playing, xv, 281
general problem solving, 257
general purpose GPU, 338
general purpose GPU computing, 182
general purpose graphic processing unit, 339
general purpose programming on the GPU, 348
general sorting algorithm, 49
generalized intelligence, vii
generalized PTSP, 390
generalized TSP, 388, 393
genetic algorithm, 104, 128, 278
geo-location data, 433
geodetic coordinate system, 166
geometric constraint, 434
geometric localization structure, 178
geometric travel planning, 178
GGP, 281
glassbox for trump game, 223
glassbox solver, 217, 218, 233
global alignment, 431
global memory, 185
global optimization, 382
global positioning system, 165
GNN, 185
Go, 67
Golomb code, 348
good assortment, 433
Google, xiii
governance, 429
GP^{2}U programming, 348
GPGPU, 182, 185, 338
GPGPU programming, 348
GPS, xv, 165
GPTSP, 390
GPU, 67, 185, 193, 201, 213, 337
GPU architecture, 186, 338
GPU Bucketsort, 213
GPU kernel, 343
GPU memory access, 340
GPU memory manager, 340
GPU memory model, 185
gradient ascent, 397
gradient descent, 369
grand, 218
grand game, 224
graph coloring, 107
graph matching problem, 305
graph neural network, 185
graph optimization, 110
graph search engineering, 388
graphical network, 191
graphics processing unit, 67, 185, 194, 337
grass-firing algorithm, 399, 409
greatest common divisor, 9
greedy best-first search, 83
greedy bisimulation, 260
greedy search, 19
greedy selection, 278
grid graph, 36
grid sampling, 379
grounding of planning problem, 271
groupage traffic, 361
GRU network, 184
GTSP, 388, 393

HA, 393
HAKMEM algorithm, 158, 164
Hala-Tafl, 215
halting problem, 14
Hamiltonian path problem, 393
Hamming distance, 155, 312
hand, 217
hand game, 226
handling of incomplete knowledge, 130
handling of sparse knowledge, 130
hard pattern matching, 130
has-a relation, 314
hash compaction, 348
hash function, 181, 195
hashing with binomial coefficients, 200
Heapsort, 50, 65
heavy-tail distribution, 117
Heller-Heller-Gorfine distance, 311
Heron's algorithm, 8
heuristic, 18, 260
heuristic method, 330
heuristic search, 18
heuristic search planning, 257
heuristic value, 207
heuristically guided depth-first branch-and-bound search, 373
Hex, 104
HGG, 311
hidden vector, 74
hierarchical memory structure, 340
high diversity, 110
high-card theorem, 221
high-confidence card, 236
high-dimensional XOR problem, 161
Hilbert curve, 153
Hilbert index, 155
Hilbert's curve, 4, 154, 155, 163
Hilbert-Schmidt information criterion, 311
Hinge loss, 154
hitting set, 111
Hoare's partition algorithm, 58
Hoeffding's independent test, 311
Holt-Winter prediction, 133
holy grail, 359
home automation, 432
hope card, 224, 236
Horn clause, 279
Horn CNF, 272
Horn theory, 280
host IDS, 130
HSF-Spin, 427
HSIC, 311
hub, 419
human intelligence, vii
human-adequate intelligence, xii
human-adequate playing strength, 254
Hungarian algorithm, 365, 393, 409
hybrid classification algorithm, 212
hybrid filtering, 310
hybrid intrusion detection, 151

hybrid intrusion detection system, 131
hybrid machine learning, 136
hyper-geometric distribution, 233
hyperspace, 310
hypothesis, 257, 258
hypothesis computation, 272
hypothesis generation, 258
hypothesis pool, 145

IBM study, 437
IDS, 129
IEEE floating point value, 46
IIS, 145
illegal state, 279
image, 153, 209, 285
image-to-image, 164
imbalanced data, 152
imitation learning, 67, 76
imperative kernel, 3
imperative programming, 3
imperfect-information game, 218
implementation simplicity, 373
implicit minimax, 104
implicit pattern database, 278
implicit PDB, 260
improved Douglas-Peuker algorithm, 178
improved iterative scaling, 145, 152
improvement of detection accuracy, 130
in-degree, 174
in-place sorting, 49, 50
in-situ probing mechanism, 269
in-situ sorting, 49
incident matrix, 145
incorporation of expert knowledge, 130
increased utility, 418
incremental depth-first branch-and-bound, 393
incremental map generation, 165
incremental map learning, 166
incremental progress measure, 287
indel, 349
indirectness, 397
indoor logistics, xiv
induction, xv
industrial practice, 386
inference process, 278
information safety, 129
information security, 129
information-theoretic optimum, 348
initial alignment, 359
input feature, 142
insert, 37
inspection for object defects, 388
inspection of drone images, 436
inspection point, 387, 399
inspection problem, 387, 401, 407
inspection quality, 401
integer linear programming, 104
integer programming, 409
integrated production and delivery, 434
Intel Haswell CPU, 84
intelligent agent, 325
intelligent intralogistics, 436
intelligent packaging robot, 362
interactive game, xiv, 401
interior detection, 381
Internet of Things, 432
intertwined logical and temporal constraint, 434
interval tree, 121
Introsort, 65
intrusion detection system, 129
inventory list, 433
inverse kinematics, ix
inversible hash function, 195
inversion, 195
inverted abstract space, 207
invertible perfect hashing, 194
IoT, 432
irregular shaped object packing, 377
is-a relation, 314
is-empty, 37
ISGD, 182, 187
iso-oriented clustering problem, 312
ISPA, 238
IT compliance, 129
IT infrastructure threats, 129
IT monitoring, 129
iteration, 72, 368
iterative gradient descent, 181
iterative machine learning, 433
iterative stochastic gradient descent, 182, 187
iterative-deepening dynamic programming, 350, 359
itinerary, 305
Iverson bracket, 80

Jacobi iteration method for solving linear equations, 339
JADE framework, 308
Java, 3
job shop scheduling, 427
joint node representation, 37, 301
Journal of Artificial Intelligence Research, 437
Julia set, 4
Jupyter Notebook, 431

Kalman filtering, 430
Karp-Steel patching, 365
KDD CUP 1999 dataset, 151
Kendall correlation coefficient, 311
Keras framework, 77
kernel function, 154, 339
kernel method, 164
kernel trick, 431
killer card, 224
killer heuritic, 296
Kinback's hand strength system, 238
Klondike Solitaire, 104
Klondike solitaire, 375
knowledge base, 217, 279
knowledge elicitation, 219
knowledge reasoning, 225, 226, 432
knowledge representation, viii, 225, 226, 432
knowledge set, 225
knowledge-based recommendation, 322
Knuth-Morris-Pratt algorithm, 132
Koch's curve, 4
Kriegspiel, 284

L'Attaque, 294
label, 188
label-correcting algorithm, 46
label-setting algorithm, 46
labeled data, 430
Lagrangian optimization, ix
laminate manufacturing, 377
lane-precise map, 178
large neighborhood search, 104
Las-Vegas randomized search, 26
last good reply, 104
last-mile connectivity, 297
LASVM, 164
latitude, 167
lava, 397
lazy greedy search, 269
LCM, 9
learning curve, 373
least common multiplier, 9
legal and regulary implication, 255
Lehmer's algorithm, 26
length of a gap, 349
level, 72, 368
level policy, 91
Levenshtein distance, 431
lexicographic rank, 196
lexicographic ranking, 215
LIBLINEAR, 164
LIBSVM, 164
likelihood, 145
likelihood ratio, 145
limited resource, 325
linear arrangement, 260
linear function, 224
linear kernel, 161
linear programming, 409
linear separation, 430
linear temporal logic, 428
linear-time ranking, 193
linear-time unranking, 193
linearly non-separable, 158
linearly separable, 158
linked decision tree, 318
List BDDA*, 263
literature snowballing, 437
LLN, 377
LM-cut heuristic, 278
load balancing, 343
local alignment, 431
local memory, 185
local search, 397
local virtual time, 425
locality, 391
locality of the search, 207
lock-free hash table, 104
logic-based abduction, 280
logical theory, 271
Lomuto's partition algorithm, 58
Long short-term memory network, 184
longitude, 167
loss function, 67, 183, 431
low-cost plan, 258
low-cost positioning system, 165
low-level compare-and-swap, 358
lower bound, 420
LSTM network, 184
LTL, 340, 428
LTL model checking, 348
Lua, 3, 68

M&S heuristic, 257, 260
machine intelligence, 429
machine learning, viii, 119, 181, 322, 406
machine translation, 184
macro action, 390, 397
macro blocking, 127
makespan, 325
malicious attack, 435
Manhattan distance, 19, 311, 399
manifestation, 271
manual event, 120
manual inspection, 119
many-core CPU, xvi
many-core model checking, 348
many-core processor, 337
many-to-many problem, 183
many-to-one problem, 183
MAP, 348
map, 391, 402
map as undirected graph, 329
map construction and refinement, 178
map generation, xv, 165
map viewer, 168, 169
Marias, 218
Markov decision process, 104, 334
mask, 170
mass confusion, 309
mass customization, 309, 322
MAST, 104
Mastermind, 28
MAT, 172
match, 144
Math Kernel Library, 84

mathematical program, 361
Mathlab, 3
matix factorization, 431
Matrix BDDA*, 263
matrix factorization, 188, 322, 436
matrix product, 430
matrix-vector multiplication, 339
max-min problem, 366
maximally sparse, 173
maximally sparse skeleton, 177
Maximum Clique, 107
maximum information coefficient, 311
maximum likelihood estimate, 159
maximum variation, 430
maze, 15
MCBSS, 291
MCS, 85, 86
MCTS, 334
MDP, 104, 334
MeanShift, 312
media industry, 127
medial axis, 378, 399, 407
medial axis transformation, 172
median, 12, 157
median selection, 164
meeting salesman problem, 434
memoization, 6
memory layers, 338
memory-limited search, 213
Memory, 294
merge-and-shrink heuristic, 257, 260, 278
MergeInsertion, 63
Mergesort, xiv, 62
merging state spaces, 260
meta block, 169
metric, 431
Metric TSP, 393
MIC, 311
MIC-*k*-means, 311
microscopic inspection, 436
microscopic traffic simulation, 177
Microsoft, xiii
middle hand, 225
middlegame, 217
min-max fusion, 138
min-max problem, 12
minibatch, 187
minigame glassbox solver, 233
minigame search, 254, 255
minigame solver, 217, 230
minimal perfect hash function, 195, 208
minimal perfect hashing, 193, 216
minimal-cost plan, 272
minimax, 281, 283
minimax score, 226
minimax search, 295
minimax strategy, 326
minimial counterexample, 428
minimin strategy, 326
minimum confidence, 315
Minimum Graph Coloring, 107
minimum support, 315
Minkowski sum, 391
misère game variant, 230
mismatch, 144
miss rate, 430
missing value problem, 433
Mister X, 27
misuse detection, 129, 150
mixed Gaussian, 164
mixed Gaussian distribution, 157
mixed strategy, 326
mixed-integer linear-problem solving, 434
mixture of Gaussian, 159
MKL, 84
MLNN, 67, 68
MLNN expressivity, 68
MO-PTSP, 388, 390
model, 182
model (to be learnt), 181
model checking, xvi, 213, 412, 428
model counting, 277
monitored learning, 430
monitoring software, 130
monolithic transition relation, 285
monotone cost function, 39
monotone heuristic, 78
monrail, 416
Monte-Carlo belief state search, 291
Monte-Carlo planning, 104
Monte-Carlo randomized search, 26
Monte-Carlo sampling, 255
Monte-Carlo search, xiv, xv, 85, 86, 334, 351, 397
Monte-Carlo Tree Search, 361
Monty Hall, 294
morphological closing, 171
morphological filter, 170
morphological opening, 170, 171
Morpion Solitaire, 72, 90, 104
most salient part extraction, 178
motion planning, 401
move, 72, 110
move history, 224
move ordering, 235
move-alteration property, 195
moving average, 133, 433
moving-test driver, 231
moving-test-driver, 225
MPNet, 389
MSA encoding, 351
MSA problem, 350
multi-arm bandit problem, 85
multi-billion-dollar industry, 255
multi-clustered problem, 312
multi-core CPU, xvi
multi-core model checking, 348
multi-dimensional packing problem, 374
multi-goal motion planning, xv, 434
multi-layer feed-forward neural network, 67
multi-objective PTSP, 390
Multi-Pivot Quicksort, 57, 66
multi-player game, 217
multi-threading support, 201
multi-valued Logic, xii
multi-vehicle transport, 299
multiagent planning, 334
multiagent simulation, xv, 298
multiagent simulation system, 297
multiagent system, viii, 298, 361
multiagent-based simulation, 297
multiarmed bandit, 103
multimedia, xv
multimedia news adaptation, 434
multimodal mobility, 297
multinomial distribution, 219
multiobjective PTSP, 388
multiple fault analysis, 278
multiple hard disks, 200
multiple PDBs, 278
multiple sequence alignment, 349
multiple sequence alignment problem, xvi
multiple-child representation, 38
multivariate Gaussian, 159
multivariate time series, 128
multiway merging, 42
multiway tree representation, 38
mutex constraint, 279
mutual exclusion, 271
Myrvold and Ruskey permutation ranking, 197

naïve Bayes model, 129
naive statistics, 126
Nash equilibrium, 255, 326
native c-code, 428
native c-code verification, 412
native GPU instruction, 338
nearest neighbor search, viii, 153, 159
nearest neighbor search problem, 154
Nebula, 311
needle in the haystack, 116
needle-in-the-haystack problem, 84
Needleham-Wunsch algorithm, 431
neighborhood local search, 409
Nested Monte-Carlo Search, 85, 86, 104, 368, 397
nested Monte-Carlo search, xv, 375
nested rollout, 85, 104
Nested Rollout Policy Adaptation, 72, 85, 86, 104, 368, 397
netcat, 168
Netflix, xiii
Netflix price, 182
network input file, 69
network intrusion detection system, 150
network output file, 69
network security, xiv
Neumann's theorem, 326
neural network, viii, xiv, 67, 69, 157, 191, 433
neural network intrusion detection, 151
neuro-symbolic approach, 432
newcomer progression, 241
Nim, 29, 289
Nine-Men's-Morris, 194, 200, 208, 215
NMC, 397
NMCS, 86, 90
NMEA 0183, 167
NN, 69
node process, 417
non-convex optimization problem, 187
non-cooperating task, 334
non-cooperative planning, 334
non-deterministic choice, 418
non-deterministic progress, 414
non-linear classification, 156
non-player character, 401
nonlinear structure, 430
nontrivial explanation, 272
normal threat, 144
normal-form game, 330
normalization, 198
NP-complete, 110
NP-hard problem, 19
NP-hardness, 108
NP-hardness of TSP, 388
NRPA, 72, 86, 368, 397
null, 218
null game, 219
number of trick, 233
number partitioning problem, 19

object orientation, 3, 362
object placement, 369
object stability, 362
object-oriented programming, 33
objective function, 415
objects first, 3, 33
observation, 290
observer pattern, 167
obstacle, 402
obstacle avoidance, 434
Occam's razor, 272, 277
OCR scan, 433
octile grid, 41
octree, 377
off-line learning, 315
off-line optimization problem, 375
on-line learning, 315
once temporal relation, 145
one-bit reachability, 195
one-player game, 193
one-to-many problem, 183
online optimization problem, 375
online play, 237
ontological background knowledge, 150

ontological knowledge, 314
ontological reasoning, viii
ontology, xv, 150
OODBC database, 167
open card game, 221
Open list, 111, 195, 259, 343
open list, 207
open-card solving, 231
open-world assumption, 275
Operations Research, 104, 434
opponent, 218
opponent model, 224
opponent paranoia search, 228
opponent strength, 240
OPTICS, 312
optimal clustered tour, 401
optimal decision-making algorithm, 361
optimal mixed strategy, 333
optimal MSA, 350
optimal packing, 362
optimal plan, 258
optimization via model checking, 412
ordered packing, 375
ordinary feed-forward neural network, 67
oriented box, 363
original state space, 208
orthogonal hash functions, 195
orthonomal matrix, 430
ouput sensitive algorithm, 121
out-degree, 174
outdoor logistics, xiv
outlier detection, 167
over-the-table play, 237
overall objective function, 415
overfitting, 145, 160, 183, 188, 431
Oware, 215
OWCTY, 348

package unloading, 375
PacMan, 31
page rank algorithm, 436
pairing heap, 37, 38
pairwise distance table, 394
pairwise sequence alignment, 349
palindrome, 5
Pancake Problem, 104, 194, 198, 215
parallel agent execution, 299
parallel BFS, 425
parallel event, 125
parallel NRPA, 359
parallel processing, 166
parallelisim for the masses, 200
parallelization, 41, 358
parallelized packing, 374
paranoia search, xv, 225, 235
paranoid search, 255
parcel delivery, 362
Pareto frontier, 391
parity, 195
Parking, 259
parsing, 262
partial expansion, 359
partial information, 225
partial information game, 253
partial order, 419
partial order reduction, 278, 428
partial overservable game, 254
partial pattern database, 260, 278
partial PDB, 260
partial-information game-tree search, 253
partial-information search, 255
partial-observable search, 225
partially matching pattern, 137
partially observable game, 218
particle swarm optimization, 104
partition, 12
partitioning, 12
partonomy, 314
Pascal's triangle, 9
path, 415
pathfinding, 409
pattern, 121
pattern collection, 278
pattern database, 35, 260
pattern database compression, 207
pattern database heuristic, 257
pattern length estimation, 133
pattern recognition, 128, 154, 163
PCA, 183, 430
PCAP, 132
PCP, 14
PCTL logic, 339
PDB, 260
PDDL, 108, 258
PDDL fragment, 273
Peano's curve, 4
Pearson correlation coefficient, 311
Peg Solitaire, 22, 194, 196, 199, 200, 215
perceptron, 67, 431
perfect hash function, xv, 195
perfect hashing, 22, 193, 194
perfect heuristic, 260
perfect-information Monte-Carlo, 248
perfect-information Monte-Carlo sampling, 253, 256
perfect-information paranoid search, 255
perfectly matching feature, 143
performance analyis, 414
performance indicator, 298
perimeter PDB, 268
perimeter search, 268
periodogram, 133
permutation game, 194
physical traveling salesman problem, 387, 408
pick-up and backhaul, 362
pickup-and-delivery problem, 375
picture, 153
pixel elimination, 173
plan, 258, 415
plan execution, 300
plan generation, 300
plan interdiction game, 335
plan recognition, xv, 257, 258
plan recognition problem, 279
plane, 67, 69
planning task, 258
planning task abstraction, 260
planning task structure, 271
player collaboration, 219
playing agreement, 219
playing convention, 226
playout, 72
Poker, 217
policy, xiv, 349, 351, 359
policy adaptation, 368, 388, 397
policy iteration, 191
policy network, xiv, 67, 68, 84
policy-based benchmarking, 115
polynomial-time algorithm, 19
pop, 39
position of a gap, 349
positive observation literal, 280
positive/negative predictive value, 430
possible assignment, 275
Post's Correspondence Problem, 14
PostgreSQL, 306
power spectral density estimation, 133
pre-classification, 430
pre-image, 285
precision, 119, 125, 430, 437
precompiled network, 433
predicate sees, 283
predicted performance, 185
predictive analytics, 437
predictive maintenance, 436
preimage, 209, 272
premium service constraint, 362
previous temporal relation, 145
primality testing, 33
primary event, 412
prime factorization, 10
principal component analysis, 128, 158, 183, 430, 433
principle of composition, 158
principle of differentiation, 158
prior knowledge, 104
prioritization, 145
priority queue, xiv, 35, 37, 402, 412, 426
PRISM, 339
prisoners' dilemma, 326
probabilistic completeness, 409
probabilistic model checking, 339
probabilistic road map, 434
probabilistically complete, 97, 116
probability of winning, 218
probe, 121
problem domain description language, 108, 258
problem solving, viii
process synchronization, 421
product component, 314
product configuration, xiv, 309, 313, 322
product of state spaces, 260
product utility, 315
production unit, 434
programming, xiv
progress measure, 286
progressive widening, 104
project team, 436
projection, 354
Promela, 417
proof-number search, 221
property parameter, 313
property validation, 428
protection of privacy, 432
protein alphabet, 349
PSDE, 133
pseudo-random number generator, 26
pthreads, 201
PTSP, 387
public-key cryptographic system, 26
publish-subscribe system, 167
pure strategy, 326
push, 39
puzzle solving, xiv
Python, 3, 431

QBF solving, ix
QC, 119
quality assurance, 119, 127
quality control, 119
Queens Problem, 16
queuing theory, 427
QuickMergesort, 62
QuickMergeXsort, 63
Quickselect, 12
Quicksort, xiv, 12, 57
QuickXsort, 59

radial basis network, 157
radix heap, 35, 37, 40, 46, 392, 399, 409
Rails, 33
random grid, 36
random MSA, 351
random prime number test, 26
random sampling, 104
randomized algorithm, 26, 33
randomized search, 33
randomness in the deal, 237
range tree, 376
rank, 193
ranking, 22, 194, 209
ranking procedure, 209
rapid action-value estimator, 104
rapidly exploring random tree, 388

raster map, 170
rational agents, xii
RAVE, 104
ray casting, 407
RBF, 153
RBF kernal, 312
RBF kernel, 163
reachable state, 208
readout, 185
real-time dynamic programming, 191
real-time execution model, 390
rear hand, 225
recall, 119, 125, 430
receiver operating characteristic curve, 431
recommendation, 316
recommender system, 309, 322, 435
recource hunting domain, 328
rectangle packing, 376
rectangle packing problem, 362
Rectified Linear Unit, 68
recurrent neural network, 183
recursion, 4
recursive Monte-Carlo search, 256
reduced ordered binary decision diagram, 209
refined ELO system, 237
regression loss, 83
regularization, 145, 183, 431
regularization technique, 152
reinforcement learning, 191, 374
relational Markov model, 279
relevance, 125
reliability, 432
ReLU, 68
removal of redundant nodes, 174
reopening, 46, 391
replanning, 300
restricted problem, 325
result filtering, 119
retail trade, 433
retrograde analysis, 67, 69, 193
retrograde analysis on a bitvector, 210
reverse Polish notation, 337, 341
reweighting, 35
RFID, 416
ride sharing, 308
RMM, 279
RNA, 349
RNN, 183
road centerline, 172
ROBDD, 209
robot packing system, 362
robotics, 406
robust packing, 377, 380
ROC curve, 431
role random, 283
rollout, xiv, 72, 85, 349, 368, 369, 398
root mean square error, 188
root parallelization, 104
Roshambo, 84
roulette wheel, 91
routing vehicle, 176
RRT, 388
RRT*, 388
RSA, 26
RSME, 188
Rubik's cube, 208, 215
Ruby, 33
rule generalization, 435
rule induction, 429
rule learning, 191
rule mining, xv, 309
rule sees, 284
run, 415

safe card, 219
safe hand, 219
safe suit, 219
SameGame, 72
sample matrix, 144
sampling based planning, 409
sampling-based motion planning, 434
Sarrus' rule, 397
SAS$^+$ planning, 258, 260
SAT, 107, 110
SAT benchmark library, 115
SAT encoding, 109
SAT solving, 117
satcount, 209
satisfaction, 137
satisfiability problem, 110
satisfying assignment, 275
SAX, 132
Scale Invariant Feature Transformation, 154
scale-free learning, 75
scanline flood-fill, 391
scenario, 402
scheduling, 412, 427
Scheme, 33
score, 110, 391
scoring, 122
scoring matix, 359
Scratch, 3
Scrum, 33
search framework, 110
search stack, 374
search tree map, 41
seasonal-trend decomposition procedure based on Loess, 134
secondary event, 412
security game, 331
security information and event management, xv, 129
security management, 129
Seeger-Fabian evaluation, 240
Seeger-Fabian system, 238
sees rule, 290
sekeleton, 172
selection games, 194
selective policy, 111
selectivity, 430
self-explnatory system, 432
semi-external model checking, 347
sensitive-based clustering, 431
sensitivity, 430
sensor fusion, 145
sentiment analysis, 183
sequence alignment, 349
sequence prediction problem, 184
sequential chaining, 348
sequential data, 128
service robotics, 436
set of expanded state, 259
set of generated state, 259
set-based game definition, 282
shallow neural network, 83
shape of image, 170
shared external space, 347
shared hyperedge, 159
shared memory, 185
shared RAM, 339
shareholder value, viii
sharp ten, 236
Sheep and Wolf, 289
shelve filling, 436
shortest paths search, 391
shrinking, 260
SIEM, xv, 129
SIEM correlation, 150
SIEM correlation engine, 151
Sieve of Eratostenes, 26
SIFT, 154, 162, 164
sigmoid-layer, 184
SIMD, 201
similarity measure, 138
simple behavior, 298
simple moving average, 133
simplex algorithm, ix
simulated annealing, 377, 382
simulation controller hierarchy, 308
simulation progress, 298
simultaneous localization and mapping, 154, 164
single instruction multiple data, 201
single-dummy miniglassbox solver, 231
single-dummy solver, 217
single-frame fingerprint, 125
single-source shortest paths, 35, 297, 388
single-source shortest-paths search, 301, 392
singular value decomposition, 430, 433
Skat, 217
Skat AI, 218
skeletal algorithm, 399
skeleton, 409
skeletonization algorithm, 173
skeletonizing technique, 178
skikit-learn library, 431
skill level, 218
skill recommender system, 436
SL, 377
SLAM, 154, 164
slice, 121
sliding-tile puzzle, 19, 194, 197, 202, 207, 215
smart home, 432
smart house, 432
smart mobility, 297
smart retail, 436
smoothed statistics, 133
smooting factor, 133
SMT solving, ix
Snake, 30
Snake-in-the-Box, 100, 104
Snort, 150
social actor, 107
social grouping, 107
social network, xv
social network analysis, 107
social network community, 115
soft pattern matching, 136
software model checking, xvi
software verification, xiv
Sokoban, 74
Soldier model, 428
sole hope card, 236
solid-state disk, 347
solution reconstruction, 262, 421
solvability status, 197
solving a game, 193
some hope card, 236
sorting, xiv
sound algorithm, 227
SP-UCT, 117
space-filling curve, 153
spam filter, 431
SPARQL, 137
sparse data problem, 435
sparse matrix-vector multiplication, 339
SPDB, 257
Spearman correlation coefficient, 311
specificity, 430
spectral clustering, 312
speech recognition, 184
speed-up, 347
speed-up technique, 46
Speeded Up Robust Features, 154
speedup, 159
sphere packing, 380
sphere tree, 377
sphere tree construction, 378, 383
Spin, 412, 417, 427
square packing, 373
SRAM, 339
SSSP, 35, 388, 392
stability, 386
stabilization, 381
Stackelberg game, 334, 335

Stackleberg game, 331
standing card, 219, 224
start fingerprint, 125
state, 391
state evaluation, 383
state of compromise, 129
state reconstruction, 197
state set flushing, 201
state space explortation, 337
state space graph, 337, 427
state space tree, 427
state variable, 412
state-explosion problem, 72
state-space exploration task, xvi
state-space search, 18, 72, 193
static analyzer, 271
static card group information, 230
static code analysis, 435
static dictionary, 348
static environment, 325
station, 414
stationary markets setup, 436
statistical feature, 124
statistical learning theory, 83
statistical machine learning, 154
statistical sampling, 255
statistical tests, 224
statistically sampled belief space, 255
Stegen's hand strength system, 238
stem product generation, 435
step-minimal explanation, 272, 273
stereolithography, 377
STL, 134
stochastic gradient descent, xv, 67, 68
Stratego, 294, 296
strategy fusion, 256
stratified selection time prediction, 268
streamed processing, 338
streaming multiprocessor, 338
streaming processor, 338
strengthened heap, 50
strip packing problem, 363, 373
strong AI, 429
strong heap, 50, 51
strong NP-hardness, 393
strong solution, 194
strong solution (to a game), 281
StrongHeapsort, 50
strongly solving a game, 193
structured configuration, 314
structured product configuration, 309
structuring element, 170
STSP, 361
subgame, 327
success likelihood, 116
successor generation on the GPU, 347
succinct state-space representation, 193
Sudoku, 17
sufficient statistics, 142
suit, 218
suit card, 219
suit factorization, 255
suit game, 219, 224
SUMO, 165
superposition, 122
SuperScalarSampleSort, 57, 65
supertrace, 425
supervised learning, 182
supervised machine learning, 119, 156
supervised statistical machine learning, 188
suport vector machine, xv
support, 315
support bridge, 363
support in 3D printing, 381
support vector machine, viii, 153, 154, 156, 163, 181, 182, 312, 431
SURF, 154, 164
surface inspection, 388
suspicious threat, 144
SVD, 430
SVM, 153, 154, 182
SVM training, 157
swap bodies, 362
swarm intelligence, xii
swarm tree search, 425
sweep-line algorithm, 174
sweep-line paradigm, 177
SymBA*, 262
Symbolic A*, 261
symbolic abduction, 258
symbolic aggregate approximation, 132
symbolic AI, 432
symbolic bidirectional search, 259
symbolic breadth-first-search, 209
symbolic model checking, 216
symbolic pattern database, 257, 268
symbolic planner, 428
symbolic planning, 257, 278
symbolic search, 284
symbolic unidirectional search, 259
symmetric TSP, 361
symmetry, 194, 206
symmetry detection, 428
synchronous product of state spaces, 260
syntax tree, 341
synthetic experiment, 160
system dynamics, 411
system of linear equation, 339
system of linear equations, 339
system state, 412

table card, 225
table lookup value, 224
tabu list, 277
Tabu Search, 118
tanh-layer, 184
task, 182
task planning, 401
taxonomy, 314
temporal difference learning, 86
temporal pattern mining, 432
temporal planning, 334
tensor, 68
Tensorflow, 77
terminal value, 313
test set, 430
text-to-image, 164
texture processing cluster, 338
TicTacToe, 68, 200, 285
Tidybot, 259
tie-breaking, 83
tile block, 169
tile manager, 167
tiled world, 166
time delay, 422
time series, 119, 128, 133
time series analysis, 132
time series prediction, 433
time series-based anomaly detection, 134
time stamp, 133
time window, 133
time-accuracy trade-off, 163
time-dependent event, 128
time-dependent street network, 306
time-expanded street network, 306
time-series prediction, 436
timed automata, 427
timeout, 422
Tolerant Pattern Matching, 129, 130, 136
tolerant pattern matching, viii, 150
top, 39
top heap, 54
top-spin puzzle, 194, 198
torch, 68
tour, 399
tourist traveler problem, 306
touristic industry, 434
Towers-of-Hanoi, 15

TPM, 136
tracability, 429
trace filter, 166, 167
trace map, 166
trace processor, 166, 167, 169
trace server, 166, 167
traceability, 429
tracking algorithm, 174
tracking flow of people, 434
traffic management, 308
traffic management plan, 297
traffic simulation, 176
training, 183, 436
training example, 430
training set, 430
transaction data, 435
transition relation, 261, 262, 271, 285
transitive closure, 274
transparency, 429, 432
transport logistics, 361
transport request, 299
transposition table pruning, 226
trapezoidal map, 392
travel time, 415
traveling salesman problem, 24, 46, 186, 361
traveling salesman problem with time windows, 86, 375
tree parallelization, 104
trend removal, 436
triangulated model, 377
triangulation, 392
tribe, 436
trick-taking, 217
trivial heuristic, 420
true game value, 69
true/false negative, 430
true/false negative rate, 430
true/false positive, 430
true/false positive rate, 430
trump card, 219
trump game, 219
truncated depth-first branch-and-bound, 393
trust, 429
trusted AI system, 429
TSAD, 134
TSP, 24, 186, 361, 428
TSP solver, 390
TSP with pickups and deliveries, 104
TSP with time windows, 91
TSPTW, 86, 91, 375
TunedQuicksort, 66
turn, 233
turnover, 433
two-bit breadth-first search, 195, 209
two-bit retrograde analysis, 196, 201
two-player game, 193
two-player turn-taking game, 285
two-player zero-sum game, 236
type I/II error, 430

UAV, 329
UCB bandit algorithm, 267
UCT, 86, 103, 104, 281
UCT player, 283
Ulam's function, 8
Ulti, 218
Ultimate Heapsort, 50, 65
undecidability, 14
under/overestimation, 430
uniform-cost inference, 272
unit propagation, 110
unloading sequence, 362
unmanned aerial vehicle, 329
unpacking, 362
unrank, 193
unranking, 22, 194, 209
unranking procedure, 209
unreachable state, 208
unspecific, 144

unsupervised learning, 128, 182, 431
update phase, 168
Upper Confidence Bounds Applied to Trees, 86
urban mobility, 297
urban mobility planning, 165
urban mobility simulation, 178
user-defined evaluation function, 381
utility, 417
utility function, 326
UTM transformation, 169

vacancy simulation, 432
valid hypothesis, 274
valid packing, 377, 386
value iteration, 191
value network, xiv, 67, 68
value of game, 326
value prediction, 133
vanishing gradient, 76
vanishing gradient problem, 183
variance, 311
vectorization, 176
vehicle controller, 401
Vehicle Routing, 104
viable candidate filtering, 400
video editing, xiv
video RAM, 186
virtual standing card, 219
virtual vehicle, 176
virus scan, 132
visibility score, 400
visible area, 399
visited flag, 369
Voronoi diagram, 378, 409
Voronoi edge, 378
Voronoi face, 378
vote, 318
voxel, 45
VRAM, 186

waiting time, 415
Walmart, xiii
ward hierarchical clustering, 312
warehouse, 434
wasted space lower bound, 373
watch literal, 117
watchman route problem, 409
waypoint finding, 409
weak AI, 429
WeakHeapsort, 65
weight vector, 432
weighted abduction, 280
weighted graph, 417
Weighted MCBSS, 293
weighted moving average, 133
Weltensuche, 235
wide neural network, 83
winning feature, 224
winning parameter, 224
winning probability, 224
winning set, 196
winning streak, 240
word-level bit operation, 24
workspace decomposition, 409
world, 225
world model, 298
world search, 235
world-wide hierarchical navigation, 169
world-wide live ELO list, 237
world-wide web, 33
worst-case search, 225
worst-case world search, 235

XML parser, 176
XOR problem, 158

Z2 model, 418
zero-cost action, 274
zero-sum game, 199, 218, 326
zoom factor, 169

References

1. A. Abdo, S. Edelkamp, and M. Lawo. Nested rollout policy adaptation for optimizing vehicle selection in complex vrps. In *41st IEEE Conference on Local Computer Networks Workshops, LCN Workshops 2016, Dubai, United Arab Emirates, November 7-10, 2016*, pages 213–221. IEEE Computer Society, 2016.
2. ACM. KDD Cup 1999: Data. Special Interest Group on Knowledge Discovery and Data Mining.
3. R. Adamski, T. Grel, M. Klimek, and H. Michalewski. Atari games and intel processors. In *IJCAI-Computer Games Workshop*, 2017.
4. A. Aggarwal, H. Edelsbrunner, P. Raghavan, and P. Tiwari. Optimal time bounds for some proximity problems in the plane. *Inf. Process. Lett.*, 42(1):55–60, 1992.
5. C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the Surprising Behavior of Distance Metrics in High Dimensional Space. In *Int. Conf. on Database Theory*, pages 420–434, 2001.
6. R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD*, pages 207–216, 1993.
7. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
8. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.
9. K. B. Akhilesh, T. G. Sitharam, M. Goswami, and D. Manjula. User Needs Study: Living Lab on Bangalore Mobility and ICT Research for Smart City Solutions. Technical report, Center for Infrastructure, Sustainable Transportation, and Urban Planning (CiSTUP), 2012.
10. S. G. Akl and M. M. Newborn. The principal continuation and the killer heuristic. In *Proceedings of theAnnual ACM Conference*, pages 466–473, 1977.
11. M. H. Albert, J. P. Grossman, R. J. Nowakowski, and D. Wolfe. An introduction to clobber. *INTEGERS: The Electronic Journal of Combinatorial Number Theory*, 5(2), 2005.
12. D. W. Albrecht, I. Zukerman, A. E. Nicholson, and A. Bud. Towards a Bayesian model for keyhole plan recognition in large domains. In *International Conference on User Modelling*, pages 365–376, 1997.
13. J. D. Allen. A note on the computer solution of connect-four. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pages 134–135. Ellis Horwood, 1989.
14. J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.
15. C. Anderson and D. Weld. Relational markov models and their application to adaptive web navigation. In *International Conference on Knowledge Discovery and Data Mining*, pages 143–152, 2002.
16. J. P. Anderson. *Computer Security Threat Monitoring and Surveillance*. James P. Andrsson Co., 1980.
17. K. Anderson, J. Schaeffer, and R. C. Holte. Partial pattern databases. In *SARA*, pages 20–34, 2007.
18. R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, Inc., second edition, 2008.
19. T. Andreas, M. Jahrer, R. M. Bell, and F. Park. The BigChaos solution to the Netflix Grand Prize. *Most*, 14(3):1–52, 2009.
20. D. E. Appelt and M. E. Pollack. Weighted abduction for plan ascription. In *User Modeling and User-Adapted Interaction, 2(1-2)*, pages 1–25, 1992.
21. C. Applegate, C. Elsaesser, and J. C. Sanborn. An architecture for adversarial planning. *IEEE Trans. Systems, Man, and Cybernetics*, 20(1):186–194, 1990.
22. I. ArcSight. ESM 101 – Concepts for ArcSight ESM, 2010.
23. I. Ardiyanto and M. J. Toyohashi. Visibility-based viewpoint planning for guard robot using skeletonization and geodesic motion model. In *IEEE International Conference on Robotics and Automation*, pages 660–666, 2013.
24. N. Ascheuer, M. Fischetti, and M. Grötschel. Solving the asymmetric travelling salesman problem with time windows by branch-and-cut. *Math. Programming*, 90:475–506, 2001.
25. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.
26. M. Aumüller and M. Dietzfelbinger. Optimal partitioning for dual pivot quicksort. In *ICALP*, pages 33–44, 2013.
27. M. Aumüller, M. Dietzfelbinger, and P. Klaue. How good is multi-pivot quicksort? *CoRR*, abs/1510.04676, 2015.
28. M. Aumüller and N. Hass. Simple and fast blockquicksort using lomuto's partitioning scheme. In S. G. Kobourov and H. Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 15–26. SIAM, 2019.
29. C. Bäckström and B. Nebel. Abductive planning with event calculus. In *International Conference and Symposium on Logic Programming*, pages 562–579, 1988.
30. T. P. Bagchi. *Multiobjective Scheduling by Genetic Algorithms*. Kluwer Academic Publishers, 1999.
31. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.
32. H. Baier and P. Drake. The power of forgetting: Improving the Last-Good-Reply policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303–309, 2010.
33. M. W. Barley, S. Franco, and P. J. Riddle. Overcoming the utility problem in heuristic generation: Why time matters. In *ICAPS*, 2014.
34. J. Barnat, L. Brim, M. Ceska, and T. Lamr. CUDA accelerated LTL Model Checking. In *The Fifteenth International Conference on Parallel and Distributed Systems (ICPADS)*, 2009.

S. Edelkamp, *Algorithmic Intelligence*, https://doi.org/10.1007/978-3-319-65596-3

35. J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In *TACAS*, volume 4963 of *LNCS*, pages 48–62, 2008.
36. H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.
37. I. Batal, L. Sacchi, R. Bellazzi, and M. Hauskrecht. Multivariate Time Series Classification with Temporal Abstractions. *Florida Artificial Intelligence Research Society Conference*, pages 344–349, 2009.
38. G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *Sigkdd Explorations*, 6:2004, 2004.
39. H. Bay, A. Ess, T. Tuytelaars, and L. J. V. Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008.
40. A. Behzad and M. Modarres. A new efficient transformation of the generalized traveling salesman problem into traveling salesman problem. In *International Conference on Systems Engineering*, page 68, 2002.
41. F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*, volume 5. Wiley, 2007.
42. R. Bellman. *Dynamic Programming*. Princton University Press, 1957.
43. R. Bent and P. V. Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers & Operations Research*, 33(4):875–893, 2006.
44. J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.
45. E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning ways*. Academic Press, 1982.
46. P. Bertoli, M. Bozzano, and A. Cimatti. A symbolic model checking framework for safety analysis, diagnosis, and synthesis. In *MoChArt*, pages 1–18, 2006.
47. K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When Is "Nearest Neighbor" Meaningful? In *Int. Conf. on Database Theory*, pages 217–235, 1999.
48. U. Bhat. Finite capacity assembly-like queues. *Queueing Systems*, 1:85–101, 1986.
49. T. C. Biedl, E. D. Demaine, M. L. Demaine, R. Fleischer, L. Jacobsen, and J. I. Munro. The complexity of clickomania. *CoRR*, cs.CC/0107031, 2001.
50. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
51. M. Birn, M. Holtgrewe, P. Sanders, and J. Singler. Simple and fast nearest neighbor search. In *Workshop on Algorithm Engineering and Experiments*, pages 43–54, 2010.
52. E. E. Bischoff and M. D. Marriott. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research*, 44:267–276, 1990.
53. R. Bjarnason, A. Fern, and P. Tadepalli. Lower bounding klondike solitaire with monte-carlo planning. In *ICAPS*, 2009.
54. Y. Björnsson and H. Finnsson. CadiaPlayer: A simulation-based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.
55. X. Blanvillain. Oware is strongly solved. In *Computer and Games*, 2022.
56. B. Bloom. Space/time trade-offs in hashing coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
57. H. Blum. *Models for the Perception of Speech and Visual Forms*. MIT Press, 1967.
58. H. Blum. A transformation for extracting new descriptors of shape. In *Models for the Perception of Speech and Visual Form*, pages 362–380, 1967.
59. M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Comput. System Sci.*, 7:448–461, 1973.
60. H.-J. Boeckenhauer and D. Bongartz. *Algorithmic Aspects of Bioinformatics*. Springer, 2010.
61. B. Bonet. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*, 2008.
62. B. Bonet and H. Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.
63. A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast Kernel Classifiers with Online and Active Learning. *Journal of Machine Learning Research*, 6:1579–1619, September 2005.
64. D. Borrmann, P. J. De Rezende, C. C. De Souza, S. P. Fekete, S. Friedrichs, A. Kröller, A. Nüchter, C. Schmidt, and D. C. Tozoni. Point guards and point clouds: Solving general art gallery problems. In *ACM Symposuim on Computational Geometry*, pages 347–348, 2013.
65. D. Bosnacki and D. Dams. Integrating real time into spin: A prototype implementation. In *FORTE / PSTV*, pages 423–438, 1998.
66. D. Bosnacki, S. Edelkamp, and D. Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In *SPIN*, pages 32–49, 2009.
67. D. Bosnacki, S. Edelkamp, and D. Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In *Model Checking Software (SPIN)*, volume 5578 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2009.
68. D. Bosnacki, S. Edelkamp, D. Sulewski, and A. Wijs. Parallel probabilistic model checking on general purpose graphics processors. *Int. J. Softw. Tools Technol. Transf.*, 13(1):21–35, 2011.
69. F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *WADS*, pages 139–150, 2007.
70. F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *CIKM*, pages 653–662, 2007.
71. L. Bottou and O. Bousquet. The Tradeoffs of Large Scale Learning. In *Advances in Neural Information Processing Systems*, volume 20, pages 161–168, 2008.
72. L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. Technical report, arXiv:1606.04838, 2016.
73. B. Bouzy. An abstract procedure to compute weak Schur number lower bound. In *Computers and Games*, 2006.

74. B. Bouzy. Monte-Carlo fork search for cooperative path-finding. In *IJCAI-Workshop on Computer Games Workshop*, pages 1–15, 2013.
75. B. Bošanský, C. Kiekintveld, V. Lisý, and M. Pěchouček. An Exact Double-Oracle Algorithm for Zero-Sum Extensive-Form Games with Imperfect Information. *Journal of Artificial Intelligence Research*, 51:829–866, 2014.
76. M. Bowling, N. Burch, M. Johanson, and O. Tammelin. Heads-up limit hold'em poker is solved. *Commun. ACM*, 60(11):81–88, 2017.
77. M. Bowling, R. Jensen, and M. Veloso. A formalization of equilibria for multiagent planning. In *IJCAI*, pages 1460–1462, 2003.
78. G. Bradshaw and C. O'Sullivan. Adaptive medial-axis approximation for sphere-tree construction. *ACM Transactions On Graphics*, 23(1):1–26, 2004.
79. R. I. Brafman, C. Domshlak, Y. Engel, and M. Tennenholtz. Planning games. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.
80. U. Brandes, F. Schulz, D. Wagner, and T. Willhalm. Travel planning with self-made maps. In *ALENEX*, 2001.
81. T. M. Breyer and R. E. Korf. 1.6-bit pattern databases. In *AAAI*, pages 39–44, 2010.
82. L. Brim, I. Cerná, P. Moravec, and J. Simsa. Accepting predecessors are better than back edges in distributed LTL model-checking. In *FMCAD*, volume 3312 of *LNCS*, pages 352–366, 2004.
83. E. Brinksma and A. Mader. Verification and optimization of a PLC control schedule. In *SPIN*, volume 1885, pages 73–92, 2000.
84. F. Brizzolari, I. Melatti, E. Tronci, and G. D. Penna. Disk based software verification via bounded model checking. In *14th Asia-Pacific Software Engineering Conference (APSEC)*, pages 358–365. IEEE Computer Society, 2007.
85. G. S. Brodal. Worst-case efficient priority queues. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58. ACM/SIAM, 1996.
86. C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2004.
87. R. Bruntrup, S. Edelkamp, S. Jabbar, and B. Scholz. Incremental map generation with GPS traces. In *IEEE Intelligent Transportation Systems*, pages 574–579, 2005.
88. A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen. Policy-based benchmarking of weak heaps and their relatives. In *SEA*, volume 6049 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 2010.
89. R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, pages 688–694, 1985.
90. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
91. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
92. R. Burke. The adaptive web. In *Hybrid Web Recommender Systems*, pages 377–408. Springer, 2007.
93. M. Burman. *New results in flow line analysis*. PhD thesis, Massachusetts Institute of Technology, 1995.
94. M. Buro, J. R. Long, T. Furtak, and N. R. Sturtevant. Improving state evaluation, inference, and search in trick-based card games. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1407–1413, 2009.
95. T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson. The computational complexity of abduction. *Artificial Intelligence*, 49(1-3):25–60, 1991.
96. A. Cabot and R. Hannum. Poker: Public policy, law, mathematics, and the future of an american tradition. *TM Cooley L. Rev.*, 22(443), 2005.
97. C. Camerer and D. Lovallo. Overconfidence and excess entry: An experimental approach. *American Economic Review*, 89(1):306–318, 1999.
98. M. Campbell, J. A. J. Hoane, and F. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.
99. S. Carlsson. A variant of Heapsort with almost optimal number of comparisons. *Information Processing Letters*, 24(4):247–250, 1987.
100. S. Carlsson. An optimal algorithm for deleting the root of a heap. *Inform. Process. Lett.*, 37(2):117–120, 1991.
101. S. Carlsson, J. Chen, and C. Mattsson. Heaps with bits. *Theoret. Comput. Sci.*, 164(1–2):1–12, 1996.
102. A. Carpenter. A CUDA implementation of support vector classification and regression, 2009.
103. M. Casey. Why Media Preservation Can't Wait: the Gathering Storm. *IASA Journal*, 44, 2015.
104. B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML*, pages 104–111. ACM, 2008.
105. T. Cazenave. Nested Monte-Carlo search. In *IJCAI*, pages 456–461, 2009.
106. T. Cazenave. Monte-Carlo beam search. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):68–72, March 2012.
107. T. Cazenave and N. Jouandeau. On the parallelization of uct. In *Computer Games Workshop (CGW)*, 2007.
108. T. Cazenave and F. Teytaud. Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In *Learning and Intelligent Optimization (LION)*, pages 42–54. Springer, 2012.
109. T. Cazenave and F. Teytaud. Beam nested rollout policy adaptation. In *ECAI-Workshop on Computer Games*, pages 1–12, 2012.
110. T. Cazenave and V. Ventos. The $\alpha\mu$ search algorithm for the game of bridge. *CoRR*, abs/1911.07960, 2019.
111. Center for Discrete Mathematics and Theoretical Computer Science. Clique and coloring problems graph format, 1993.

112. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. Technical Report 07-017, Department of Computer Science and Engineering - University of Minnesota, 2007.
113. G.-B. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.
114. N. V. Chawla, N. Japkowicz, and A. Kotcz. Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explor. Newsl.*, 6:1–6, 2004.
115. G. Chen. A gentle tutorial of recurrent neural network with error backpropagation. *arXiv*, 1610.02583, 2018.
116. S. F. Chen and R. Rosenfeld. A gaussian prior for smoothing maximum entropy models. Technical Report CMV-CS-99-108, Carnegie Mellon University, 1999.
117. T. Chen and S. Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71(1–3):269–295, 1996.
118. B. V. Cherkassy, A. V. Goldberg, and T. Ratzig. Shortest path algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1997.
119. L. Chrestien, T. Pevný, A. Komenda, and S. Edelkamp. Heuristic search planning with deep neural networks using imitation, attention and curriculum learning. *CoRR*, abs/2112.01918, 2021.
120. L. Chrestien, T. Pevný, A. Komenda, and S. Edelkamp. A differentiable loss function for learning heuristics in A. *CoRR*, abs/2209.05206, 2022.
121. L. Chrpa, J. Gemrot, and M. Pilát. Towards a safer planning and execution concept. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 972–976, 2017.
122. L. Chrpa, P. Rytír, R. Horcík, J. Cuhel, A. Livochka, and S. Edelkamp. Adversary strategy sampling for effective plan generation. In *SOCS*, pages 164–166. AAAI Press, 2021.
123. L. Chrpa, P. Rytír, R. Horcík, and S. Edelkamp. Competing for resources: Estimating adversary strategy for effective plan generation. In *AAAI*, pages 9707–9715. AAAI Press, 2022.
124. L. Chrpa, P. Rytír, A. Nyporko, R. Horcík, and S. Edelkamp. Effective planning in resource-competition problems by task decomposition. In *SOCS*, pages 47–55. AAAI Press, 2022.
125. L. Chrpa, M. Vallati, and L. McCluskey. The 2014 International Planning Competition. Description of participating planners, 2014.
126. A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *ECP*, LNCS, pages 130–142, 1997.
127. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artif. Intell.*, 147(1-2):35–84, 2003.
128. A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI*, pages 875–881, 1998.
129. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
130. J. Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139, 2007.
131. R. Coester, A. Gustavsson, T. Olsson, and A. Rudstroem. Enhancing web-based configuration with recommendations and cluster-based help. In *Workshop on Recommendation and Personalization in eCommerce*, 2002.
132. W. W. Cohen. Fast effective rule induction. In *International Conference on Machine Learning*. Morgan Kaufmann, 1995.
133. G. Cohensius, R. Meir, N. Oved, and R. Stern. Bidding in spades. In *ECAI*, pages 387–394, 2020.
134. A. Coles, M. Fox, D. Long, and A. Smith. Planning with problems requiring temporal coordination. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, pages 892–897, 2008.
135. G. Cooperman and L. Finkelstein. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37/38:95–118, 1992.
136. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, 3nd edition, 2009.
137. R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pages 72–83, 2006.
138. R. Coulom. Computing "Elo ratings" of move patterns in the game of Go. *ICGA Journal*, 30(4):199–208, 2007.
139. T. G. Crainic, G. Perboli, and R. Tadei. Extreme point-based heuristics for three-dimensional bin packing. *INFORMS Journal on Computing*, 20(3):368–384, 2008.
140. T. G. Crainic, G. Perboli, and R. Tadei. *Recent Advances in Multi-Dimensional Packing Problems*. InTech, 2012.
141. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.
142. R. Croson, P. Fishman, and D. Pope. Poker superstars: Skill or luck? similarities between golf - thought to be a game of skill - and poker. *Chance*, 21(4):25–28, 2008.
143. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.
144. S. Cunningham and P. de Nier. File-based Production: Making It Work In Practice. *International Broadcasting Convention*, 2, 2007.
145. X. Dang, L. Chrpa, and S. Edelkamp. Deep RRT*. In *SOCS*, pages 333–335. AAAI Press, 2022.
146. T. Danner and L. E. Kavraki. Randomized planning for short inspection paths. In *IEEE International Conference on Robotics and Automation*, pages 971–976, 2000.
147. A. Darwiche. Compiling knowledge into decomposable negation normal form. In *IJCAI*, pages 284–289, 1999.
148. A. Dasen. AIS: A system for the Automatic Interpretation of Street maps. *in German)*. Master's thesis, Philosophische und Naturwissenschaftliche Universität zu Bern, 2001.

149. M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In *Atlas of Protein Sequence and Structure*, volume 5, chapter 22, pages 345–352. National Biomedical Research Foundation, 1978.
150. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3 edition, 2008.
151. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2 edition, 2000.
152. T. Dean and M. Wellman. *Planning and Control*. Morgan Kaufmann Publishers, 1990.
153. M. A. DeDonno and D. K. Detterman. Poker is a skill. *Gaming Law Review*, 12(1):31–36, 2008.
154. D. E. Denning. An intrusion-detection model. In *IEEE Transactions on Software Engineering*, 1986.
155. O. Depren, M. Topallar, E. Anarim, and M. K. Ciliz. An intelligent intrusion detection system (ids) for anomaly and misuse detection in computer networks. In *Expert Systems with Applications*, volume 29, pages 713–722, 2005.
156. O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *Symposium on Computational Geometry*, pages 106–114, 2001.
157. J. E. Dickerson and J. Dickerson. Fuzzy network profiling for intrusion detection. In *Fuzzy Information Processing Society, 2000. NAFIPS. 19th International Conference of the North American*, pages 301 –306, 2000.
158. T. G. Dietterich. Machine learning for sequential data: A review. In *Structural, syntactic, and statistical pattern recognition*, pages 15–30. Springer, 2002.
159. M. Dietzfelbinger and S. Edelkamp. Perfect hashing for state spaces in BDD representation. In *KI*, volume 5803 of *Lecture Notes in Computer Science*, pages 33–40. Springer, 2009.
160. M. Dietzfelbinger and S. Edelkamp. Perfect hashing for state spaces in BDD representation. In *KI*, pages 33–40, 2009.
161. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
162. J. F. Dillenburg and P. C. Nelson. Perimeter search. *Artif. Intell.*, 65(1):165–178, 1994.
163. V. Dimitrijevic and Z. Saric. An efficient transformation of the generalized traveling salesman problem into the traveling salesman problem on digraphs. *Information Science*, 102(1–4):105–110, 1997.
164. K. Dorer and M. Calisti. An adaptive solution to dynamic transport optimization. In *AAMAS*, pages 45–51, 2005.
165. D. H. Douglas and T. K. Peuker. Algorithms for the reduction of the number of points. *The Canadian Cartographer*, 10:112–122, 1973.
166. W. B. Dowsland. Three dimensional packing solution approaches and heuristic development. *International Journal of Operational Research*, 29:1637–1685, 1991.
167. K. Dräger, B. Finkbeiner, and A. Podelski. Directed model checking with distance-preserving abstractions. In *SPIN*, pages 19–36, 2006.
168. K. Dräger, B. Finkbeiner, and A. Podelski. Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer*, 11(1):27–37, 2009.
169. M. Dreef and B. Borm, P. andvan der Genugten. On strategy and relative skill in poker. *International Game Theory Review*, 5(2):83–103, 2003.
170. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, 1988.
171. M. Drozdzynski, S. Edelkamp, A. Gaubatz, S. Jabbar, and M. Liebe. On constructing a base map for collaborative map generation and its application in urban mobility planning. In *ITSC*, pages 678–683. IEEE, 2007.
172. P. Duersch, M. Lambrecht, and J. Oechsler. Measuring skill and chance in games. *European Economic Review*, 127, 2020.
173. Y. Dumas, J. Desrosiers, E. Gelinas, and M. Solomon. An optimal algorithm for the travelling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.
174. G. H. Dunteman. *Principal components analysis*. Number 69 in Quantitative Applications in the Social Sciences. Sage, 1989.
175. R. D. Dutton. Weak-heap sort. *BIT*, 33:372–381, 1993.
176. M. Dweighter. Problem e2569. *American Mathematical Monthly*, 82:1010, 1975.
177. W. Echelmeyer, A. Kirchheim, A. L. Lilienthal, H. Akbiyik, and M. Bonini. Performance indicators for robotics systems in logistics applications. In *IROS Workshop on Metrics and Methodologies for Autonomous Robot Teams in Logistics (MMART-LOG)*, 2011.
178. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, pages 13–24, 2001.
179. S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *AIPS*, pages 274–293, 2002.
180. S. Edelkamp. Automated creation of pattern database search heuristics. In *MOCHART*, pages 36–51, 2007.
181. S. Edelkamp. Deep or wide? learning policy and value neural networks for combinatorial games. In *Workshop on Computer Games, CGW, Revised Selected Papers*, volume 705 of *Communications in Computer and Information Science*, pages 19–33, 2016.
182. S. Edelkamp. Improving the cache-efficiency of shortest path search. In *KI*, volume 10505, pages 99–113. Springer, 2017.
183. S. Edelkamp. Challenging human supremacy in skat. In P. Surynek and W. Yeoh, editors, *SOCS*, pages 52–60. AAAI Press, 2019.
184. S. Edelkamp. Dynamic play via suit factorization search in Skat. In *KI*, volume 12325 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2020.
185. S. Edelkamp. Representing and reducing uncertainty for enumerating the belief space to improve endgame play in Skat. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 395–402. IOS Press, 2020.
186. S. Edelkamp. Improving computer play in skat with hope cards. In *Computer and Games (CG)*, 2023.
187. S. Edelkamp and T. Cazenave. Improved diversity in nested rollout policy adaptation. In *KI*, volume 9904 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2016.

188. S. Edelkamp, A. Elmasry, and J. Katajainen. The weak-heap data structure: Variants and applications. *Journal of Discrete Algorithms*, 16:187 – 205, 2012.
189. S. Edelkamp, A. Elmasry, and J. Katajainen. Heap construction - 50 years later. *Comput. J.*, 60(5):657–674, 2017.
190. S. Edelkamp, A. Elmasry, and J. Katajainen. Optimizing binary heaps. *Theory Comput. Syst.*, 61(2):606–636, 2017.
191. S. Edelkamp, E. Externest, S. Kühl, and S. Kuske. Solving graph optimization problems in a framework for monte-carlo search. In A. Fukunaga and A. Kishimoto, editors, *SOCS*, pages 163–164. AAAI Press, 2017.
192. S. Edelkamp, T. Federholzner, and P. Kissmann. Searching with partial belief states in general games with incomplete information. In *KI*, volume 7526 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2012.
193. S. Edelkamp and M. Gath. Pickup-and-delivery problems with time windows and capacity constraints using nested Monte-Carlo search. In *ICAART*, 2014.
194. S. Edelkamp and M. Gath. Solving single vehicle pickup and delivery problems with time windows and capacity constraints using nested monte-carlo search. In *ICAART*, pages 22–33, 2014.
195. S. Edelkamp, M. Gath, T. Cazenave, and F. Teytaud. Algorithm and knowledge engineering for the TSPTW problem. In *IEEE Symposium Series on Computational Intelligence (SSCI)*, 2013.
196. S. Edelkamp, M. Gath, C. Greulich, M. Humann, O. Herzog, and M. Lawo. Monte-carlo tree search for logistics. In U. Clausen, H. Friedrich, C. Thaller, and C. Geiger, editors, *Commercial Transport*, pages 427–440. Springer International Publishing, 2016.
197. S. Edelkamp, M. Gath, and M. Rohde. Monte-carlo tree search for 3D packing with object orientation. In *German Conference on Artificial Intelligence*, pages 285–296, 2014.
198. S. Edelkamp and C. Greulich. Solving physical traveling salesman problems with policy adaptation. In *IEEE Conference on Computational Intelligence and Games, CIG*, pages 1–8, 2014.
199. S. Edelkamp and C. Greulich. Using SPIN for the optimized scheduling of discrete event systems in manufacturing. In *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, volume 9641 of *Lecture Notes in Computer Science*, pages 57–77. Springer, 2016.
200. S. Edelkamp and C. Greulich. Nested rollout policy adaptation for multiagent system optimization in manufacturing. In H. J. van den Herik, A. P. Rocha, and J. Filipe, editors, *ICAART (1)*, pages 284–290. SciTePress, 2017.
201. S. Edelkamp and C. Greulich. A case study of planning for smart factories. *Int. J. Softw. Tools Technol. Transf.*, 20(5):515–528, 2018.
202. S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *Model Checking Software, 13th International SPIN Workshop*, volume 3925 of *LNCS*, pages 1–18, 2006.
203. S. Edelkamp, S. Jabbar, and T. Willhalm. Geometric travel planning. *IEEE Transactions on Intelligent Transportation Systems*, 6(1):5–16, 2005.
204. S. Edelkamp and F. Jacob. Learning event time series for the automated quality control of videos. In *KI*, volume 9904 of *Lecture Notes in Computer Science*, pages 148–154. Springer, 2016.
205. S. Edelkamp and P. Kissmann. Externalizing the multiple sequence alignment problem with affine gap costs. In *KI*, volume 4667 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 2007.
206. S. Edelkamp and P. Kissmann. Limits and possibilities of BDDs in state space search. In *AAAI*, pages 1452–1453, 2008.
207. S. Edelkamp and P. Kissmann. Symbolic classification of general two-player games. In *KI*, pages 185–192, 2008.
208. S. Edelkamp and P. Kissmann. Optimal symbolic planning with action costs and preferences. In *IJCAI*, pages 1690–1695, 2009.
209. S. Edelkamp and P. Kissmann. On the complexity of BDDs for state space search: A case study in Connect Four. In *AAAI*. AAAI Press, 2011.
210. S. Edelkamp, P. Kissmann, and M. Rohte. Symbolic and explicit search hybrid through perfect hash functions – A case study in Connect Four. In *ICAPS*. AAAI, 2014.
211. S. Edelkamp, P. Kissmann, D. Sulewski, and H. Messerschmidt. Finding the needle in the haystack with heuristically guided swarm tree search. In *MKWI*, pages 2295–2308. Universitätsverlag Göttingen, 2010.
212. S. Edelkamp, P. Kissmann, and Á. Torralba. Lex-partitioning: A new option for BDD search. In *GRAPHITE*, 2012.
213. S. Edelkamp, P. Kissmann, and Á. Torralba. Symbolic A* search with pattern databases and the merge-and-shrink abstraction. In *ECAI*, pages 306–311, 2012.
214. S. Edelkamp, P. Kissmann, and Á. Torralba. Bdds strike back (in AI planning). In *AAAI*, pages 4320–4321. AAAI Press, 2015.
215. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *SPIN*, pages 57–79, 2001.
216. S. Edelkamp, M. Pomarlan, and E. Plaku. Multiregion inspection by combining clustered traveling salesman tours with sampling-based motion planning. *IEEE Robotics Autom. Lett.*, 2(2):428–435, 2017.
217. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *KI*, pages 81–92, 1999.
218. S. Edelkamp, P. Sanders, and P. Simecek. Semi-external LTL model checking. In *CAV*, pages 530–542, 2008.
219. S. Edelkamp and S. Schrödl. Localizing A*. In *AAAI*, pages 885–890, 2000.
220. S. Edelkamp and S. Schrödl. Route planning and map inference with global positioning traces. In *Computer Science in Perspective, Essays Dedicated to Thomas Ottmann*, volume 2598 of *Lecture Notes in Computer Science*, pages 128–151. Springer, 2003.
221. S. Edelkamp and S. Schrödl. *Heuristic Search: Theory and Applications*. Morgan Kaufmann, 2012.
222. S. Edelkamp, B. C. Secim, and E. Plaku. Surface inspection via hitting sets and multi-goal motion planning. In *TAROS*, volume 10454, pages 134–149. Springer, 2017.

223. S. Edelkamp and M. Stommel. The bitvector machine: A fast and robust machine learning algorithm for non-linear problems. In *ECML/PKDD (I)*, volume 7523 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2012.
224. S. Edelkamp and D. Sulewski. Flash-efficient LTL model checking with minimal counterexamples. In *SEFM*, pages 73–82, 2008.
225. S. Edelkamp and D. Sulewski. Model checking via delayed duplicate detection on the GPU. Technical Report 821, Technische Universität Dortmund, 2008. Presented at 22nd Workshop on Planning, Scheduling, and Design PUK.
226. S. Edelkamp and D. Sulewski. Efficient explicit-state model checking on general purpose graphics processors. In *Model Checking Software (SPIN)*, volume 6349 of *Lecture Notes in Computer Science*, pages 106–123. Springer, 2010.
227. S. Edelkamp and D. Sulewski. Perfect hashing for domain-dependent planning on the GPU. In *International Conference on Automated Planning and Scheduling*, 2010.
228. S. Edelkamp, D. Sulewski, and C. Yücel. Perfect hashing for state space exploration on the GPU. In *ICAPS*, pages 57–64. AAAI, 2010.
229. S. Edelkamp and Z. Tang. Monte-carlo tree search for the multiple sequence alignment problem. In *SOCS*, pages 9–17. AAAI Press, 2015.
230. S. Edelkamp and A. Weiß. QuickXsort: Efficient sorting with n logn - 1.399n + o(n) comparisons on average. In *CSR*, pages 139–152, 2014.
231. S. Edelkamp and A. Weiß. Blockquicksort: Avoiding branch mispredictions in quicksort. *ACM J. Exp. Algorithmics*, 24(1):1.4:1–1.4:22, 2019.
232. S. Edelkamp and A. Weiß. Worst-case efficient sorting with quickmergesort. In *ALENEX*, pages 1–14. SIAM, 2019.
233. S. Edelkamp, A. Weiß, and S. Wild. Quickxsort: A fast sorting scheme in theory and practice. *Algorithmica*, 82(3):509–588, 2020.
234. S. Edelkamp and P. Wichern. Packing irregular-shaped objects for 3d printing. In S. Hölldobler, M. Krötzsch, R. Peñaloza, and S. Rudolph, editors, *KI*, volume 9324 of *Lecture Notes in Computer Science*, pages 45–58. Springer, 2015.
235. R. C. Edgar. MUSCLE: a multiple sequence alignment method with high accuracy and throughput. *Nucleic acids research*, 32(5):1792–7, 2004.
236. R. C. Edgar. MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC bioinformatics*, 5(113), 2004.
237. J. Egeblad, B. K. Nielsen, and A. Odgaard. Fast neighborhood search for two- and three-dimensional nesting problems. *Europ. Journ. of Oper. Res.*, 183(3):1249–1266, 2007.
238. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42:3–42, 1995.
239. C. Elfers, S. Edelkamp, and O. Herzog. Efficient tolerant pattern matching with constraint abstractions in description logic. In J. Filipe and A. L. N. Fred, editors, *ICAART*, pages 256–261. SciTePress, 2012.
240. C. Elfers, S. Edelkamp, and H. Messerschmidt. Combining conditional random fields and background knowledge for improved cyber security. In *KI*, volume 8077, pages 284–287. Springer, 2013.
241. J. L. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993.
242. B. Englot and F. Hover. Sampling-based coverage path planning for inspection of complex structures. In *International Conference on Automated Planning and Scheduling*, pages 29–37, 2012.
243. M. Enzenberger and M. Müller. A lock-free multithreaded Monte-Carlo tree search algorithm, 2012. ACG.
244. E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo. A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. In *Applications of Data Mining in Computer Security*. Kluwer, 2002.
245. S. Evangelista. Dynamic delayed duplicate detection for external memory model checking. In *Model Checking Software, 15th International SPIN Workshop*, volume 5156 of *LNCS*, pages 77–94, 2008.
246. R. Fabbri, L. F. Estrozi, L. Da, and F. Costa. On voronoi diagrams and medial axes. *Journal of Mathematical Imaging and Vision*, 17:27–40, 2002.
247. J. Faigl, V. Vonásek, and L. Preucil. A multi-goal path planning for goal regions in the polygonal domain. In *European Conference on Mobile Robots*, pages 171–176, 2011.
248. A. Falkner and A. Felfernig. Recommendation technologies for configurable products. *AI Magazine*, 32(3):99–108, 2011.
249. W. Fan, M. Miller, S. J. Stolfo, and W. Lee. Using artificial anomalies to detect unknown and known network intrusions. In *IEEE International conference on Data Mining*, pages 123–130. IEEE Computer Society, 2001.
250. F. Fang, P. Stone, and M. Tambe. When Security Games Go Green: Designing Defender Strategies to Prevent Poaching and Illegal Fishing. In *In Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.
251. A. Felfernig and R. Burke. Constraint-based recommender systems: Technologies and research issues. In *ICEC*, volume 3, pages 1–10, 1988.
252. A. Felfernig, G. Friedrich, and L. Schmidt-Thieme. Recommender systems. *IEEE Intelligent Systems*, 22(5), 2007.
253. A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.
254. I. Fiedler and J.-P. Rock. Quantifying skill in games - theory and empirical evidence for poker. *Gaming Law Review and Economics*, 13(1):50–57, 2009.
255. H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 1134–1139, 2008.
256. H. Finnsson and Y. Björnsson. Learning Simulation Control in General-Game-Playing Agents. In *AAAI Conference on Artificial Intelligence*, 2010.
257. K. Fischer, J. P. Mueller, and M. Pischel. Cooperative transportation scheduling: An application domain for dai. *Journal of Applied Artificial Intelligence*, 10:1–33, 1995.
258. P. Flach. *Machine Learning*. Cambridge University Press, 2014.

259. R. Fleischer. A tight lower bound for the worst case of Bottom-up-heapsort. *Algorithmica*, 11(2):104–115, 1994.
260. R. Fleischer, B. Sinha, and C. Uhrig. A lower bound for the worst case of Bottom-up-heapsort. *Inform. and Comput.*, 102(3):263–279, 1993.
261. G. Fleishanderl, G. E. Friedrich, A. Haselbock, H. Schreiner, and M. Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems and their applications*, 13(4):59—-68, 1998.
262. A. Florio and F. Berrilli. A skeletonizing algorithm for granulation and supergranulation cell finding. In *Poster Proceedings of SOLE98 Workshop*, 1998.
263. R. W. Floyd. Algorithm 245: Treesort 3. *Commun. ACM*, 7(12):701, 1964.
264. J. Footen and J. Faust. *The Serive-Oriented Media Enterprise: SOA, BPM and Web Services in Professional Media Systems*. Elsevier/Focal Press, Amsterdam [u.a.], 1. edition, 2008.
265. K. D. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, 1993.
266. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.
267. S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *IEEE Symposium on Research in Security and Privacy*, pages 202–212. IEEE Computer Society Press, 1994.
268. M. Fox and D. Long. The detection and exploration of symmetry in planning problems. In *IJCAI*, pages 956–961, 1999.
269. S. Franco, Á. Torralba, L. H. S. Lelis, and M. Barley. On creating complementary pattern databases. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 4302–4309, 2017.
270. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
271. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithm. *Journal of the ACM*, 34(3):596–615, 1987.
272. T. C. Fu. A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1):164–181, 2011.
273. T. M. Furtak. *Symmetries and Search in Trick-Taking Card Games*. PhD thesis, University of Alberta, 2013.
274. A. Galuszka and A. Swierniak. Planning in multi-agent environment using strips representation and non-cooperative equilibrium strategy. *Journal of Intelligent and Robotic Systems*, 58(3-4):239–251, 2010.
275. P. Garcia-Teodoro, J. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security*, 28(1-2):18 – 28, 2009.
276. M. R. Garey, D. S. Johnson, and R. Sethi. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, 1(2):117–129, may 1976.
277. B. Gärtner. Fast and robust smallest enclosing balls. In *ESA*, pages 325–338, 1999.
278. R. Gassner. Solving Nine-Men-Morris. *Computational Intelligence*, 12:24–41, 1996.
279. P. Gastin and P. Moro. Minimal counterexample generation in SPIN. In *Model Checking Software, 14th International SPIN Workshop*, volume 4595 of *LNCS*, pages 24–38, 2007.
280. W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27:47–57, 1979.
281. M. Gath, O. Herzog, and S. Edelkamp. Agent-based planning and control for groupage traffic. In *IEEE-CEWIT*, 2013.
282. D. Gay, R. Guigoures, M. Boullé, and F. Clérot. Feature extraction over multiple representations for time series classification. In *New Frontiers in Mining Complex Patterns*, pages 18–34. Springer, 2014.
283. J. D. Gehrke, A. Schuldt, and S. Werner. Quality Criteria for Multiagent-Based Simulations with Conservative Synchronisation. In *Dedicated Conference on Simulation in Production and Logistics (ASIM)*, pages 177–186, 2008.
284. R. Geisberger and D. Schieferdecker. Heuristic contraction hierarchies with approximation guarantee. In *SOCS*, pages 31–38, 2010.
285. S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In Z. Ghahramani, editor, *Proceedings of the International Conference on Machine Learning (ICML)*, pages 273–280. ACM, 2007.
286. S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875, 2011.
287. M. R. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.
288. L. Geneste and M. Ruet. Experience-based configuration, 2001.
289. J. A. George and D. F. Robinson. A heuristic for packing boxes into a container. *Computer and Operational Research*, 7:147–156, 1980.
290. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Symposium Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.
291. L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.
292. G. Ghiani, F. Guerriero, G. Laporte, and R. Musmanno. Real-time vehicle routing: Solution concepts, algorithms and parallel computing strategies. *European Journal of Operational Research*, 151:1–11, 2003.
293. T. Giannakopoulos, a. Pikrakis, and S. Theodoridis. A novel efficient approach for audio segmentation. *2008 19th International Conference on Pattern Recognition*, pages 2–5, 2008.
294. M. Ginsberg. Step toward an expert-level Bridge-playing program. In *IJCAI*, pages 584–589, 1999.
295. F. Giunchiglia and P. Traverso. Planning as model checking. In *ECP*, pages 1–19, 1999.
296. P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, pages 176–185, 1991.
297. G. Goh, N. Cammarata, C. Voss, S. Carter, M. Petrov, L. Schubert, A. Radford, and C. Olah. Multimodal neurons in artificial neural networks. *Distill*, 2021.

298. C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24(1-2):67–100, 2000.
299. G. H. Gonnet and J. I. Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, 1986.
300. R. Gößl. *Der Skatfuchs – Gewinnen im Skatspiel mit Mathematische Methoden*. Selfpublisher. Dämmig, Chemnitz, Available from the Author or via DSKV Altenburg, 2019.
301. N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *SIGMOD*, pages 325–336, 2006.
302. S. Grandmontagne. *Meisterhaft Skat spielen*. Selfpublisher, Krüger Druck+Verlag, 2005.
303. C. Greulich and S. Edelkamp. Branch-and-bound optimization of a multiagent system for flow production using model checking. In *ICAART*, 2016.
304. C. Greulich and S. Edelkamp. Two model checking approaches to branch-and-bound optimization of a flow production system. In H. J. van den Herik and J. Filipe, editors, *ICAART 2016, Revised Selected Papers*, volume 10162 of *Lecture Notes in Computer Science*, pages 19–36, 2016.
305. C. Greulich, S. Edelkamp, and N. Eicke. Cyber-physical multiagent-simulation in production logistics. In *MATES*, volume 9433, pages 119–136. Springer, 2015.
306. C. Greulich, S. Edelkamp, and M. Gath. Agent-based multimodal transport planning in dynamic environments. In I. J. Timm and M. Thimm, editors, *KI*, volume 8077, pages 74–85. Springer, 2013.
307. C. Greulich, S. Edelkamp, M. Gath, T. Warden, M. Humann, O. Herzog, and T. G. Sitharam. Enhanced shortest path computation for multiagent-based intermodal transport planning in dynamic environments. In *ICAART (2)*, pages 324–329. SciTePress, 2013.
308. E. Groshev, M. Goldstein, A. Tamar, S. Srivastava, and P. Abbeel. Learning generalized reactive policies using deep neural networks. *arXiv preprint arXiv:1708.07280*, 2017.
309. Q. Gu, Z. Cai, L. Zhu, and B. Huang. Data mining on imbalanced data sets. In *Advanced Computer Theory and Engineering, 2008. ICACTE '08. International Conference on*, pages 1020–1024, December 2008.
310. S. Gudmundsson, T. P. Runarsson, and S. Sigurdsson. Support vector machines and dynamic time warping for time series. In *International Joint Conference on Neural Networks (IJCNN)*, pages 2772–2776, 2008.
311. A. Guenther. *Flexible Kontrolle in Expertensystemen zur Planung und Konfigurierung in technischen Domaenen*. infix, 1992.
312. R. H. Gueting and D. Wood. Finding rectangle intersections by divide-and-conquer. *IEEE Transactions on Computers*, 33(7):671–675, 1984.
313. L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.
314. K. K. Gupta, B. Nath, and K. Ramamohanarao. Conditional Random Fields for Intrusion Detection. In *International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, pages 203–208. IEEE Press, 2007.
315. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
316. M. Hall-May, J. Houpert, C. Tiensch, H. Fassold, and V. Engen. Analysis of Loss Modes in Preservation Systems. Technical Report Deliverable D2.2, DAVID Digital AV Media Damage Prevention and Repair, 2014.
317. W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.
318. J. Han. *Data Mining*. Springer, 2009.
319. P. Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High Performance Computing – HiPC 2007*, pages 197–208, 2007.
320. S. Harmel. *Skat–Zahlen*. Klabautermann-Verlag, Pünderich (Mosel), 2016.
321. M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. In *GPU Gems 3*. Addison Wesley, August 2007.
322. J. Harrison. Assembly-like queues. *Journal of Applied Probability*, 10:354–367, 1973.
323. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.
324. P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, pages 1007–1012, 2007.
325. M. Hassani, Y. Lu, J. Wischnewsky, and T. Seidl. A geometric approach for mining sequential patterns in interval-based data streams. In *2016 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2016, Vancouver, BC, Canada, July 24-29, 2016*, pages 2128–2135. IEEE, 2016.
326. M. Hatem and W. Ruml. External memory best-first search for multiple sequence alignment. In *AAAI*, 2013.
327. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
328. Y. He, W. Chen, M. Yang, and W. Peng. Ontology based cooperative intrusion detection system. In H. Jin, G. Gao, Z. Xu, and H. Chen, editors, *Network and Parallel Computing*, volume 3222 of *LNCS*, pages 419–426. 2004.
329. L. T. Heberlein, B. Mukherjee, and K. Levitt. Internetwork security monitor: An intrusion-detection system for large-scale networks. In *National Computer Security Conference*, pages 262–271, October 1992.
330. A. Helias, F. Guerrin, and J.-P. Steyer. Using timed automata and model-checking to simulate material flow in agricultural production systems – application to animal waste management. *Computers and Electronics in Agriculture*, 63(2):183–192, 2008.
331. M. Helmert. Complexity results for standard benchmark domains in planning. *Artif. Intell.*, 143(2):219–262, 2003.

332. M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 2009.
333. M. Helmert, P. Haslum, and J. Hoffmann. Explicit-state abstraction: A new method for generating heuristic functions. In *AAAI*, pages 1547–1550, 2008.
334. S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *National Academy of Sciences*, 89(22):10915–10919, November 1992.
335. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. The arrow distributed directory protocol. In *SPIN*, pages 235–239, 2003.
336. J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst*, 22(1):5–53, 2004.
337. J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peuker algorithm for line simplification. *ACM Computational Geometry*, pages 383–384, 1994.
338. D. S. Hirschberg. A linear space algorithm for computing common subsequences. *Commun. ACM*, 18(6):341–343, 1975.
339. J. R. Hobbs, M. Stickel, D. Appelt, and P. Martin. Interpretation as abduction. *Artificial Intelligence*, 63:69–142, 1990.
340. J. Hochberg, K. Jackson, C. Stallings, J. McClary, D. DuBois, and J. Ford. Nadir: An automated system for detecting network intrusion and misuse. *Computers & Security*, pages 235–248, 1993.
341. S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.
342. J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001.
343. O. Hollmann, T. Wagner, and A. Guenter. Engcon: A flexible domain-independent configuration engine. In *ECAI–Workshop on Configuration*, pages 94–96, 2000.
344. R. C. Holte, A. Felner, J. Newton, R. Meshulam, and D. Furcy. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, 170(16–17):1123–1136, 2006.
345. G. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.
346. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.
347. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
348. W. Hopp and J. Simon. Bounds and heuristics for assembly-like queues. *Queueing Systems*, 4:137–156, 1989.
349. R. Horcík, Á. Torralba, P. Rytír, L. Chrpa, and S. Edelkamp. Optimal mixed strategies for cost-adversarial planning games. In *ICAPS*, pages 160–168. AAAI Press, 2022.
350. E. Hordern. *Sliding Piece Puzzles*. Oxford University Press, 1986.
351. E. Huang and R. E. Korf. New improvements in optimal rectangle packing. In *IJCAI*, pages 511–516, 2009.
352. E. Huang and R. E. Korf. Optimal rectangle packing on non-square benchmarks. In *AAAI*, 2010.
353. E. Huang and R. E. Korf. Optimal packing of high-precision rectangles. In *The International Symposium on Combinatorial Search (SOCS)*, 2011.
354. S.-C. Huang, B. Arneson, R. B. Hayward, M. Mueller, and J. Pawlewicz. Mohex 2.0: A pattern-based MCTS Hex player. In *Computers and Games*, pages 60–71, 2013.
355. P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.
356. C. Huffman and B. E. Kahn. Variety for sale: Mass customization or mass confusion? *Journal of retailing*, 74(4):491–513, 1998.
357. K. Hwang, Y. Chen, S. Member, and M. Qin. Hybrid intrusion detection with weighted signature generation over anomalous internet episodes. *IEEE Trans. on Dependable and Secure Computing*, 4(1):41–55, 2007.
358. D. hyeon Lee, D. young Kim, and J. il Jung. Multi-stage intrusion detection system using hidden Markov model algorithm. In *Proceedings of the 2008 International Conference on Information Science and Security*, ICISS, pages 72–77. IEEE Computer Society, 2008.
359. L. Hüther, B. J. Berger, S. Edelkamp, S. Eken, L. Luhrmann, H. Rothe, M.-S. Schröder, and K. Sohr. Machine learning in the context of static application security testing – ML-SAST. In *KI*, Studies for the German Federal Office for Information Security (BSI), 2022.
360. I. Ikonen, W. E. Biles, A. Kumar, J. C. Wissel, and R. K. Ragade. A genetic algorithm for packing three-dimensional non-convex objects having cavities and holes. In *International Conference on Genetic Algorithms*, pages 591–598, 1997.
361. S. S. Intille and A. F. Bobick. A framework for recognizing multi-agent action from visual evidence. In *AAAI*, pages 518–525, 1999.
362. S. Jabbar. GPS-based navigation in static and dynamic environments. Master's thesis, Universität Freiburg, 2003.
363. S. Jabbar. *External Memory Algorithms for State Space Exploration in Model Checking and Planning*. PhD thesis, University of Dortmund, 2008.
364. M. Jain, V. Conitzer, and M. Tambe. Security Scheduling for Real-world Networks. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 215–222, 2013.
365. G. Jaychandran, V. Vishal, and V. S. Pande. Using massively parallel simulations and Markovian models to study protein folding: examining the Villin head-piece. *J. Chem. Phys.*, 124(6):164 903–164 914, 2006.
366. R. M. Jensen, M. M. Veloso, and M. H. Bowling. Obdd-based optimistic and strong cyclic adversarial planning. In *ECP*, 2001.
367. R. M. Jensen, M. M. Veloso, and R. E. Bryant. State-set branching: Leveraging BDDs for heuristic search. *Artif. Intell.*, 172(2–3):103–139, 2008.

368. D. S. Johnson and L. A. McGeoch. Experimental analysis of heuristics for the STSP. In G. Gutin and A. P. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 369–443. Kluwer Academic Publishers, 2002.
369. D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the Held-Karp traveling salesman bound. In *SODA*, pages 341–350, 1996.
370. D. T. Jones, W. R. Taylor, and J. M. Thornton. The rapid generation of mutation data matrices from protein sequences. *Comput. Appl. Biosci.*, 8(3):275–282, June 1992.
371. R. Jonker and A. Volgenant. Improving the hungarian assignment algorithm. *Operations Research Letters*, 5:171–175, 1986.
372. S. K. Jonnalagadda and S. S. Mallela. An intelligent hybrid structure for improving intrusion detection. *International Journal of Research and Reviews in Software Engineering (IJRRSE)*, 1(2), 2011.
373. A. Jonsson and M. Rovatsos. Scaling up multiagent planning: A best-response approach. In *The 21st International Conference on Automated Planning and Scheduling, ICAPS*, 2011.
374. J. Jordán and E. Onaindia. Game-theoretic approach for non-cooperative planning. In *The Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1357–1363, 2015.
375. J. Jordán, A. Torreño, M. de Weerdt, and E. Onaindia. A better-response strategy for self-interested planning agents. *Appl. Intell.*, 48(4):1020–1040, 2018.
376. A. Junghanns and J. Schaeffer. Single agent search in the presence of deadlocks. In *AAAI*, pages 419–424, 1998.
377. N. Justesen, T. Mahlmann, S. Risi, and J. Togelius. Playing multiaction adversarial games: Online evolutionary planning versus tree search. *IEEE Trans. Games*, 10(3):281–291, 2018.
378. N. Justesen and S. Risi. Continual online evolutionary planning for in-game build order adaptation in starcraft. In *The Genetic and Evolutionary Computation Conference, GECCO 2017*, pages 187–194, 2017.
379. A. Kakas and P. Mancarella. Database updates through abduction. In *Very Large Data Bases*, pages 650–661, 1990.
380. K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *ESA*, pages 780–791, 2006.
381. H. Kaplan, R. E. Tarjan, and U. Zwick. Fibonacci heaps revisited. *arXiv.org*, 1407.5750v1:187–205, 2014.
382. R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Springer US, 1972.
383. R. M. Karp and J. M. Steele. The traveling salesman problem, 1990.
384. J. Katajainen. The ultimate heapsort. In X. Lin, editor, *CATS 1998*, volume 20 of *Australian Computer Science Communications*, pages 87–95. Springer, 1998.
385. K. Kato and T. Hosino. Singular value decomposition for collaborative filtering on a gpu. *IOP Conference Series: Materials Science and Engineering*, 10(1):012017, 2010.
386. S. Kato, S. Oono, H. Seki, and H. Itoh. Cost-based abduction using binary decision diagrams. In *IAE/AIE*, pages 215–225, 1999.
387. S. Katoh. MAFFT multiple sequence alignment software version 7: improvements in performance and usability. *Molecular Biology and Evolution*, 30:772–780, 2013.
388. M. Katz and C. Domshlak. Implicit abstraction heuristics. *J. Artif. Intell. Res. (JAIR)*, 39:51–126, 2010.
389. M. Katz and J. Hoffmann. Mercury planner: Pushing the limits of partial delete relaxation. *IPC 2014 planner abstracts*, pages 43–47, 2014.
390. M. Katz, J. Hoffmann, and M. Helmert. How to relax a bisimulation? In *ICAPS*, 2012.
391. M. Katz, S. Sohrabi, H. Samulowitz, and S. Sievers. Delfi: Online planner selection for cost-optimal planning. *IPC-9 planner abstracts*, pages 57–64, 2018.
392. H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *ECAI*, pages 1194–1201, 1996.
393. L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.
394. G. D. Kazazakis and A. A. Argyros. Fast positioning of limited-visibility guards for the inspection of 2d workspaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2843–2848, 2002.
395. T. Keller. 18, 20, weg – Automatisches Reizen und Drücken beim Skat. Technical report, 2007.
396. T. Keller and S. Kupferschmid. Automatic bidding for the game of Skat. In *KI*, pages 95–102, 2008.
397. R. Khanna and H. Liu. System approach to intrusion detection using hidden markov model. In *International Conference on Wireless Communications and Mobile Computing*, IWCMC, pages 349–354. ACM, 2006.
398. W. Khreich, E. Granger, R. Sabourin, and A. Miri. Combining hidden markov models for improved anomaly detection. In *IEEE International Conference on Communications (ICC)*, pages 1–6, 2009.
399. T. Kinback. *Skat-Rätsel – 50 lehrreiche Skataufgaben mit Lösungen und Analysen*. Books on Demand, Norderstedt, 2007.
400. P. Kissmann and S. Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In *KI*, volume 5803, pages 1–8. Springer, 2009.
401. P. Kissmann and S. Edelkamp. Gamer, a general game playing agent. *Künstliche Intell.*, 25(1):49–52, 2011.
402. P. Kissmann and S. Edelkamp. Improving cost-optimal domain-independent symbolic planning. In *AAAI*, 2011.
403. P. Kissmann and J. Hoffmann. What's in it for my BDD? On causal graphs and variable orders in planning. In *ICAPS*, 2013.
404. P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.
405. C. Kloth. *Systemgestaltung im Broadcasting Engineering. Prozessorientierte Konzeption integrierter Fernsehproduktionssysteme*. Vieweg+Teubner, 1. edition, 2010.
406. K. Knott, C. P. Han, and P. J. Egbelu. A heuristic approach to the three-dimensional cargoloading problem. *Int. J. Prod. Res.*, 27(5):757–774, 1989.

407. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley Longman, Reading, 2nd edition, 1998.
408. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.
409. A. Kolobov, Mausam, and D. S. Weld. LRTDP versus UCT for online probabilistic planning. In *Proceedings of AAAI*, 2012.
410. J. Kopena and W. Regli. DAMLJessKB: A tool for reasoning with the semantic web. *IEEE Intelligent Systems*, 18(3):74–77, May/June 2003.
411. Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.
412. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
413. R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
414. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI*, pages 700–705, 1997.
415. R. E. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106:181–203, 1998.
416. R. E. Korf. Optimal rectangle packing: Initial results. In *ICAPS*, pages 287–295, 2003.
417. R. E. Korf. Optimal rectangle packing: New results. In *ICAPS*, pages 142–149, 2004.
418. R. E. Korf. Minimizing disk I/O in two-bit-breath-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 317–324, 2008.
419. R. E. Korf and T. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.
420. R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.
421. D. Krajzewicz, M. Bonert, and P. Wagner. The open source traffic simulation package SUMO. In *RoboCup 2006 Infrastructure Simulation Competition*, 2006.
422. D. Krajzewicz, G. Hertkorn, C. Roessel, and P. Wagner. SUMO (Simulation of Urban Mobility); an open-source traffic simulation. In *Proceedings of the 4th Middle East Symposium on Simulation and Modelling (MESM2002)*, pages 183–187. SCS European Publishing House, 2002.
423. T. Krebs. *Knowledge-Based Configuration*, chapter encoway, pages 271–279. Morgan Kaufmann, 2014.
424. D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms*. Discrete Mathematics and Its Applications, 1984.
425. C. Krügel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 201–208, New York, NY, USA, 2002. ACM.
426. S. Kumar and E. H. Spafford. A Software Architecture to support Misuse Intrusion Detection. In *Proceedings of the 18th National Information Security Conference*, pages 194–204, 1995.
427. D. Kunkle and G. Cooperman. Solving Rubik's Cube: disk is the new RAM. *Commun. ACM*, 51(4):31–33, 2008.
428. S. Kupferschmid. Entwicklung eines Double-Dummy Skat Solvers mit einer Anwendung für verdeckte Skatspiele. Master's thesis, Albert-Ludwigs-Universität Freiburg, 2003.
429. S. Kupferschmid and M. Helmert. A Skat player based on Monte-Carlo simulation. In *Computers and Games*, pages 135–147, 2006.
430. S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *SPIN*, pages 35–52, 2006.
431. S. Kurtz. Lecture notes on basics of sequence analysis, 2007.
432. S. Kushagra, A. López-Ortiz, A. Qiao, and J. I. Munro. Multi-pivot quicksort: Theory and experiments. In *ALENEX*, pages 47–60, 2014.
433. L. Lam, S. Lee, and C. Y. Suen. Thinning methodologies-a comprehensive survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(9):869–885, 1992.
434. P. Lamborn and E. Hansen. Layered duplicate detection in external-memory model checking. In *Model Checking Software, 15th International SPIN Workshop*, volume 5156 of *LNCS*, pages 160–175, 2008.
435. M. Lanctot, M. Winands, T. Pepels, and N. Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014*, pages 341–348, 2014.
436. M. Lanctot, V. Zambaldi, A. Gruslys, A. Lazaridou, K. Tuyls, J. Pérolat, D. Silver, and T. Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4190–4203, 2017.
437. E. Lasker. *Das verständige Kartenspiel*. August Scherl Verlag, Berlin, 1929.
438. E. Lasker. *Strategie der Spiele – Skat*. August Scherl Verlag, Berlin, 1938.
439. P. Laskov, C. Schaefer, and I. Kotenko. Intrusion detection in unlabeled data with quarter-sphere support vector machines. In *PROC. DIMVA*, pages 71–82, 2004.
440. S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
441. E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, 1976.
442. W. Lee and S. J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3:227–261, 2000.
443. N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. *CoRR*, abs/0909.5649, 2009.
444. L. H. S. Lelis, S. Franco, M. Abisrror, M. Barley, S. Zilles, and R. C. Holte. Heuristic subset selection in classical planning. In *IJCAI*, pages 3185–3191, 2016.
445. J. Letchford and Y. Vorobeychik. Optimal interdiction of attack plans. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13*, pages 199–206. IFAAMAS, 2013.
446. V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
447. D. N. L. Levy. The Million Pound Bridge program. In *Heuristic Programming in Artificial Intelligence*. 1989.

448. R. R. Lewis. *A Guide to Graph Colouring: Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2015.
449. X. Li, W. Yu, X. Lin, and S. S. Iyengar. On optimizing autonomous pipeline inspection in 3D environment. *IEEE Transactions on Robotics*, 28(1):223–233, 2012.
450. D. P. Liebana, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. M. Lucas. Solving the physical traveling salesman problem: Tree search and macro actions. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(1):31–45, 2014.
451. Y. Lien, E. Ma, and B. W. Wah. Transformation of the generalized traveling salesman problem into the standard traveling salesman problem. *Information Science*, 74:177–189, 1993.
452. A. Lim and W. Ying. A new method for the three dimensional container packing problem. In *IJCAI*, pages 342–347, 2001.
453. E. Lipper and E. Sengupta. Assembly-like queues with finite capacity: bounds, asymptotics and approximations. *Queueing Systems*, pages 67–83, 1986.
454. W. Liu, Z. Gu, J. Xu, Y. Wang, and M. Yuan. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *CODES+ISSS*, pages 61–70, 2009.
455. A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *MOCHART*, pages 77–86, 2003.
456. J. R. Long. *Search, Inference and Opponent Modelling in an Expert-Caliber Skat Player*. PhD thesis, University of Alberta, 2011.
457. M. Lorenz, C. Ober-Blöbaum, and O. Herzog. Planning for autonomous decision-making in a logistic scenario. In *Proceedings of the 21st European Conference on Modelling and Simulation*, pages 140–145, Prague, CZ, 2007.
458. F. Lorenzi and F. Ricci. *Intelligent Techniques for Web Personalization*, chapter Case-based recommender systems: A unifying view, pages 89–113. Springer, 2005.
459. N. C. Love, T. L. Hinrichs, and M. R. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, 2006.
460. D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *International Converence on Computer Vision (ICCV)*, pages 1150–1157, 1999.
461. T. F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *11th National Computer Security Conference*, 1988.
462. T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, P. G. Neumann, H. S. Javitz, A. Valdes, and T. D. Garvey. A real-time intrusion-detection expert system (IDES). Technical Report Project 6784, SRI, 1992.
463. M. Manitz. Queueing-model based analysis of assembly lines with finite buffers and general service times. *Computers & Operations Research*, 35(8):2520 – 2536, 2008.
464. S. Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. Chapman & Hall/CRC machine learning & pattern recognition series. Taylor & Francis, 2014.
465. C. Martínez, M. E. Nebel, and S. Wild. Analysis of branch misses in quicksort. In *Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 114–128, 2015.
466. C. Martínez and S. Roura. Optimal Sampling Strategies in Quicksort and Quickselect. *SIAM J. Comput.*, 31(3):683–705, 2001.
467. H. Masnadi-Shirazi, V. Mahadevan, and N. Vasconcelos. On the design of robust classifiers for computer vision. In *23rd IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.
468. B. Massey. Coloring problems dimacs graph format, 2001.
469. H. Mayer, I. Laptev, A. Baumgartner, and C. Steger. Automatic road extraction based on multi-scale modeling, context, and snakes. In *International Archives of Photogrammetry and Remote Sensing*, volume XXXII, Part 3–2W3, pages 106–113, 1997.
470. C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10(3):352–365, 1989.
471. H. B. McMahan, G. J. Gordon, and A. Blum. Planning in the Presence of Cost Functions Controlled by an Adversary. In *ICML*, pages 536–543, 2003.
472. J. Méhat and T. Cazenave. Ary, a general game playing program. In *13th Board Game Studies Colloquium*, 2010.
473. K. Mehlhorn and V. Priebe. On the all-pairs shortest-path algorithm of moffat and takaoka. *Random Struct. Algorithms*, 10(1-2):205–220, 1997.
474. K. Menschner, H. Gehring, and M. Meyer. A computer-based heuristic for packing pooled shipment containers. *European Journal of Operational Research*, 1(44):277–288, 1990.
475. U. Meyer. Average-case complexity of single-source shortest-paths algorithms:lower and upper bounds. *J. Algorithms*, 48(1):91–134, 2003.
476. J. V. Michalowicz, J. M. Nichols, and F. Bucholtz. Calculation of Differential Entropy for a Mixed Gaussian Distribution. *Entropy*, 10(3):200–206, 2008.
477. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diragram with attributed edges for efficient boolean function manipuation. In *DAC*, pages 52–57, 1990.
478. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.
479. M. D. Moffitt and M. E. Pollack. Optimal rectangle packing: A Meta-CSP approach. In *ICAPS*, pages 93–102, 2006.
480. E. Morales Kluge, F. Ganji, and B. Scholz-Reiter. Intelligent products - towards autonomous logistic processes - a work in progress paper. In *Intern. PLM Conf.*, 2010.

481. I. Moraru, S. Edelkamp, S. Franco, and M. Martínez. Simplifying automated pattern selection for planning with symbolic pattern databases. In *KI*, volume 11793, pages 249–263. Springer, 2019.
482. F. Mörchen. *Time series knowledge mining*. Citeseer, 2006.
483. C. G. Morgan. Hypothesis generation by machine. *Artificial Intelligence*, 2:179–187, 1971.
484. R. Moskovitch and Y. Shahar. Temporal Patterns Discovery from Multivariate Time Series via Temporal Abstraction and Time-Interval Mining, 2009.
485. E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations. In *Symposium on Computational Geometry*, pages 274–283, 1996.
486. C. Mues and J. Vanthienen. Improving the scalability of rule base verification. In *KI*, pages 381–395, 2004.
487. M. Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving go endgames. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 578–583, 1999.
488. D. R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, 1997.
489. W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.
490. J. Nalepa, M. Blocho, and Z. J. Czech. Co-operation schemes for the parallel memetic algorithm. In *Parallel Processing and Applied Mathematics (PPAM,I)*, pages 191–201, 2013.
491. D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004.
492. A. S. Nezhad, M. Vatani, F. Barazandeh, and A. R. Rahimi. Multi objective optimization of part orientation in stereolithography. In *International Conference on Simulation, Modelling and Optimization*, pages 36–40, 2009.
493. H. T. Ng and R. J. Mooney. On the role of coherence in abductive explanation. In *AAAI*, pages 337–342, 1990.
494. M. Nicolett and K. M. Kavanagh. Magic quadrant for security information and event management. Gartner Research document G00176034, 2010.
495. R. Nissim, J. Hoffmann, and M. Helmert. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *IJCAI*, pages 1983–1990, 2011.
496. P. Norvig. *A Unified Theory of Inference for Text Understanding*. PhD thesis, EECS Department, University of California, Berkeley, 1987.
497. M. Otte. On solving floating point SSSP using an integer priority queue. *CoRR*, abs/1606.00726, 2016.
498. A. E. Ouassouli, L. Robinault, and V. Scuturici. Mining complex temporal dependencies from heterogeneous sensor data streams. In B. C. Desai, D. Anagnostopoulos, Y. Manolopoulos, and M. Nikolaidou, editors, *Proceedings of the 23rd International Database Applications & Engineering Symposium, IDEAS 2019, Athens, Greece, June 10-12, 2019*, pages 23:1–23:10. ACM, 2019.
499. D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of hidden markov models to detecting multi-stage network attacks. In *Annual Hawaii International Conference on System Sciences*, page 10 pp., 2003.
500. N. Padhye and K. Deb. Multi-objective optimisation and multi-criteria decision making for FDM using evolutionary approaches. In *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing*, pages 219–247, 2011.
501. R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA*, pages 121–133, 2001.
502. A. Palombo, R. Stern, R. Puzis, A. Felner, S. Kiesel, and W. Ruml. Solving the snake in the box problem with heuristic search: First results. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, pages 96–104, 2015.
503. G. Papadopoulos, H. Kurniawatia, and N. M. Patrikalakis. Asymptotically optimal inspection planning using systems with differential constraints. In *IEEE International Conference on Robotics and Automation*, pages 4126–4133, 2013.
504. S. Patra, M. Ghallab, D. S. Nau, and P. Traverso. Acting and planning using operational models. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, pages 7691–7698, 2019.
505. G. Paul. Approaches to abductive reasoning - an overview. *Artificial Intelligence Review*, 7:109–152, 1993.
506. W. R. Pearson. Using the fasta program to search protein and dna sequence databases. *Methods in Molecular Biology*, 24:307–331, 1994.
507. C. S. Peirce. Abduction and induction. In J. Buchler, editor, *Philosophical Writings of Peirce*, pages 150–156. Dover Books, 1955.
508. D. Perez, E. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. Cowling, and S. Lucas. Solving the physical travelling salesman problem: Tree search and macro-actions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2013.
509. D. Perez, P. Rohlfshagen, and S. M. Lucas. The physical travelling salesman problem: WCCI 2012 competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.
510. D. Perez, S. Samothrakis, S. M. Lucas, and P. Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *GECCO*, pages 351–358, 2013.
511. H. Permuter, J. Francos, and I. H. Jermyn. A study of Gaussian mixture models of colour and texture features for image classification and segmentation. *Pattern Recognition*, 39(4):695–706, 2006.
512. J. Perolat, B. de Vylder, D. Hennes, E. Tarassov, F. Strub, V. de Boer, P. Muller, J. T. Connor, N. Burch, T. Anthony, S. McAleer, R. Elie, S. H. Cen, Z. Wang, A. Gruslys, A. Malysheva, M. Khan, S. Ozair, F. Timbers, T. Pohlen, T. Eccles, M. Rowland, M. Lanctot, J.-B. Lespiau, B. Piot, S. Omidshafiei, E. Lockhart, L. Sifre, N. Beauguerlange, R. Munos, D. Silver, S. Singh, D. Hassabis, and K. Tuyls. Mastering the game of stratego with model-free multiagent reinforcement learning. *CORR*, 2206.15378, 2022.
513. J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *J. Comp. Chem.*, 26:1781–1802, 2005.

514. V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1 – 11, 2013.
515. E. Plaku. Robot motion planning with dynamics as hybrid search. In *AAAI*, 2013.
516. J. C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.
517. D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.
518. H. E. Pople. On the mechanism of abductive logic. In *IJCAI*, pages 147–152, 1973.
519. P. Porras. Stat – a state transition analysis tool for intrusion detection. Technical report, University of California at Santa Barbara, 1993.
520. P. Porras and P. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Information Security Conference*, 1997.
521. P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. The common intrusion detection framework architecture. Technical report, University of California at Santa Barbara, 1999.
522. E. J. Powley, D. Whitehouse, and P. I. Cowling. Monte-Carlo tree search with macro-actions and heuristic route planning for the multiobjective physical travelling salesman problem. In *IEEE Conference on Computational Intelligence in Games (CIG)*, pages 73–80, 2013.
523. A. Pozanco, Y. E-Martín, S. Fernández, and D. Borrajo. Counterplanning using goal recognition and landmarks. In *The Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*, pages 4808–4814, 2018.
524. F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.
525. M. Quambusch. *Gläserne Karten – Gewinnen beim Skat*. Stomi Verlag, Schwerte Rau Verlag, Düsseldorf, 1990.
526. C. Ranze, T. Scholz, T. Wagner, A. Guenter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt. A structure based configuration tool: Drive solution designer-dsd. In *AAAI*, pages 845–852, 2002.
527. S. Rashidian, E. Plaku, and S. Edelkamp. Motion planning with rigid-body dynamics for generalized traveling salesman tours. In *Proceedings of the Seventh International Conference on Motion in Games*, pages 87–96. ACM, 2014.
528. D. Ratner and M. Warmuth. Finding a shortest solution for the n x n extension of the 15-puzzle is intractable. In *AAAI*, pages 168–172, 1986.
529. K. Ravi, R. Bloem, and F. Somenzi. A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In *FMCAD*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.
530. D. Rebstock, C. Solinas, and M. Buro. Learning policies from human data for Skat. *CoRR*, abs/1905.10907, 2019.
531. D. Rebstock, C. Solinas, M. Buro, and N. R. Sturtevant. Policy based inference in trick-taking card games. *CoRR*, abs/1905.10911, 2019.
532. A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, July 1994.
533. P. Resnick and H. R. Varian. Recommender systems. *Commun. ACM 40 (1997), 3,*, 40(3):56–58, 1997.
534. F. Ricci, L. Rokach, and B. Shapira. *Recommender Systems Handbook*, chapter Recommender systems: Introduction and challenges, pages 1–34. Springer, 2015.
535. E. Rich and K. Knight. *Artificial Intelligence*. McGraw Hill, 1991.
536. S. Richter and M. Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.
537. K. Rieck and P. Laskov. Language models for detection of unknown attacks in network traffic. *Journal in Computer Virology*, 2:243–256, 2007.
538. A. Rimmel, F. Teytaud, and T. Cazenave. Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows. *Applications of Evolutionary Computation*, pages 501–510, 2011.
539. M. Roesch and S. Telecommunications. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration (LISA)*, pages 229–238, 1999.
540. S. Rogers, P. Langley, and C. Wilson. Mining GPS data to augment road models. In *Knowledge Discovery and Data Mining (KDD)*, pages 104–113, 1999.
541. M. Rohde and W. Echelmeyer. Cooperation possibilities between research and industry: Parcelrobot. *Bremer Value Reports für Produktion und Logistik*, 3(1):1–18, 2010.
542. R. Rojas. *Neural Networks: A Systematic Introduction*. Springer, New York, NY, USA, 1996.
543. J. W. Romein and H. E. Bal. Awari is solved. *J. Int. Comput. Games Assoc.*, 25:162–165, 2002.
544. J. W. Romein and H. E. Bal. Solving Awari with parallel retrograde analysis. *Computer*, 36(10):26–33, 2003.
545. C. D. Rosin. Nested rollout policy adaptation for monte carlo tree search. In *IJCAI*, pages 649–654, 2011.
546. J. Ruan and M. Thielscher. The Epistemic Logic Behind the Game Description Language. In *AAAI Conference on Artificial Intelligence*, 2011.
547. T. C. Ruys. Optimal scheduling using branch and bound with SPIN 4.0. In *Model Checking Software, 10th International SPIN Workshop*, pages 1–17, 2003.
548. T. C. Ruys and E. Brinksma. Experience with literate programming in the modelling and validation of systems. In *TACAS*, pages 393–408, 1998.
549. J. Ryan, M. jang Lin, and R. Miikkulainen. Intrusion detection with neural networks. In *Advances in Neural Information Processing Systems*, pages 943–949. MIT Press, 1998.
550. P. Rytíř, L. Chrpa, and B. Bošanský. Using classical planning in adversarial problems. In *Proceedings of the 31st IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1327–1332, 2019.

551. D. Sabin and R. Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13:42–49, 1998.
552. P. Sanders and S. Winkel. Super scalar sample sort. In *ESA*, pages 784–796, 2004.
553. F. C. Schadd. *Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis*. PhD thesis, University of Maastricht, 2009.
554. M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J. B. Chaslot, and J. W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games*. Springer, 2008.
555. J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, and M. Müller. Solving checkers. In *IJCAI*, pages 292–297, 2005.
556. P. Schallauer, H. Fassold, A. Hofmann, W. Bailer, and S. Wechtitsch. Improving preservation and access processes of audio-visual media by content-based quality assessment. *LNCS*, 7733(PART 2):385–394, 2013.
557. R. E. Schapire and Y. Freund. A brief introduction to boosting. *IJCAI*, 2(5):1401–1406, 1999.
558. S. Schiffel and M. Thielscher. Fluxplayer: A successful general game player. In *AAAI*, pages 1191–1196, 2007.
559. S. Schiffel and M. Thielscher. Reasoning About General Games Described in GDL-II. In *AAAI Conference on Artificial Intelligence*, 2011.
560. D. Schneegaß, A. M. Schäfer, and T. Martinetz. The Intrinsic Recurrent Support Vector Machine. In *European Symposium on Artificial Neural Networks (ESANN)*, pages 325–330, 2007.
561. S. Schoelkopf. *Learning with Kernels*. MIT Press, 2001.
562. S. Schrödl. An improved search algorithm for optimal multiple sequence alignment. *Journal of Artificial Intelligence Research*, 23:587–623, 2005.
563. S. Schrödl, S. Rogers, and C. Wilson. Map refinement from GPS traces. Technical Report RTC 6/2000, DaimlerChreisler Research adn Technology North America, Palo Alto, CA, 2000.
564. A. Schuldt. *Multiagent Coordination Enabling Autonomous Logistics*. Springer, 2011.
565. A. Schuldt, J. D. Gehrke, and S. Werner. Designing a simulation middleware for fipa multiagent systems. In *IEEE/WIC/ACM Conference on Web Intelligence and Intelligent Agent Technology*, pages 109–113, 2008.
566. M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.
567. M. M. Sebring and R. A. Whitehurst. Expert systems in intrusion detection: A case study. In *National Computer Security Conference*, 1988.
568. R. Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, Oct. 1978.
569. I. Sergeev. On the asymptotic complexity of sorting. *Electron. Colloquium Comput. Complex.*, TR20-096, 2020.
570. S. Shaheen, S. Guzman, and H. Zhang. Bikesharing in Europe, the Americas, and Asia. *Transportation Research Record: Journal of the Transportation Research Board*, 2143(-1):159–167, 2010.
571. S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.
572. S. Shalev-Shwartz, O. Shamir, and S. Shammah. Failures of deep learning. *CoRR*, abs/1703.07950, 2017.
573. W. Shen, L. Wang, and Q. Hao. Agent-Based Distributed Manufacturing Process Planning and Scheduling: A State-of-the-Art Survey. *IEEE Transactions on Systems, Man and Cybernetics*, 36(4):563–577, 2006.
574. D. B. Shmoys, A. H. G. Rinnooy Kan, J. K. Lenstra, and E. L. Lawler. *The Traveling salesman problem : a guided tour of combinatorial optimization*. J. Wiley and sons, 1987.
575. F. Sievers, A. Wilm, D. Dineen, T. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Soding, J. Thompson, and D. Higgins. Fast, scalable generation of high quality protein multiple sequence alignments using Clustal Omega. *Mol. Syst. Biol.*, 7(539), 2011.
576. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.
577. D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. Technical Report 1712.018, arxiv, 2017.
578. B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986.
579. P. Skobelev. Multi-agent systems for real time resource allocation, scheduling, optimization and controlling: Industrial applications. In *HOLOMAS*, pages 1–14, 2011.
580. J. Slaney and S. Thiebaux. Blocks world revisited. *Artificial Intelligence*, 125(1):119 – 153, 2001.
581. A. Smith and M. Osborne. Regularisation techniques for conditional random fields: Parameterised versus parameter-free. In *International Joint Conference on Natural Language Processing*, pages 896–907, 2005.
582. S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. lin Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. Dids (distributed intrusion detection system) - motivation, architecture, and an early prototype. In *National Computer Security Conference*, pages 167–176, 1991.
583. C. Solinas, D. Rebstock, and M. Buro. Improving search with supervised learning in trick-based card games. *CoRR*, abs/1903.09604, 2019.
584. C. Solinas, D. Rebstock, and M. Buro. Improving search with supervised learning in trick-based card games. In *AAAI*, pages 1158–1165. AAAI Press, 2019.
585. R. M. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6(1):84–85, 1977.
586. R. M. Solovay and V. Strassen. Erratum a fast Monte-Carlo test for primality. *SIAM J. Comput.*, 7(1):118, 1978.

587. P. Speicher, M. Steinmetz, M. Backes, J. Hoffmann, and R. Künnemann. Stackelberg planning: Towards effective leader-follower state space search. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
588. S. Spiegel, J. Gaebler, A. Lommatzsch, E. D. Luca, and S. Albayrak. Pattern Recognition and Classification for Multivariate Time Series Categories and Subject Descriptors. *Time*, pages 34–42, 2011.
589. S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagl, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids - a graph based intrusion detection system for large networks. In *National Information Systems Security Conference*, pages 361–370, 1996.
590. S. Staniford-Chen, B. Tung, and D. Schnackenberg. The common intrusion detection framework (cidf). In *Information Survivability Workshop*, 1998.
591. J. T. Stasko and J. S. Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, 1987.
592. R. Stern, M. Kalech, and A. Felner. Searching for a k-clique in unknown graphs. In *SOCS*, pages 83–89, 2010.
593. U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *CAV*, pages 172–183, 1992.
594. F. Stober and A. Weiß. On the average case of MergeInsertion. *Theory Comput. Syst.*, 64(7):1197–1224, 2020.
595. F. Stober and A. Weiß. Lower bounds for sorting 16, 17, and 18 elements. *CORR*, 2206.05597, 2022.
596. M. Stommel, S. Edelkamp, T. Wiedemeyer, and M. Beetz. Fractal approximate nearest neighbour search in log-log time. In *BMVC*. BMVA Press, 2013.
597. M. Stommel and O. Herzog. Binarising SIFT-Descriptors to Reduce the Curse of Dimensionality in Histogram-Based Object Recognition. In D. Slezak, S. K. Pal, B.-H. Kang, J. Gu, H. Kurada, and T.-H. Kim, editors, *Signal Processing, Image Processing and Pattern Recognition*, pages 320–327. Springer, 2009.
598. M. Stommel, M. Langer, O. Herzog, and K.-D. Kuhnert. A Fast, Robust and Low Bit-Rate Representation for SIFT and SURF Features. In *IEEE Intern. Symposium on Safety, Security, and Rescue Robotics*, pages 278–283, 2011.
599. M. Stumptner. An overview of knowledge-based configuration. *AI Commun.*, 10(2):111–125, 1997.
600. N. R. Sturtevant. Chinese checkers bitboards for move generation and ranking using bitboards and bmi2 pext and pdep instructions. In *Computer and Games*, 2022.
601. N. R. Sturtevant and R. E. Korf. On pruning techniques for multi-player games. In *AAAI/IJCAI*, pages 201–207, 2000.
602. N. R. Sturtevant and A. M. White. Feature construction for reinforcement learning in hearts. In *Computers and Games*, pages 122–134. Springer, 2006.
603. D. Sulewski, S. Edelkamp, and P. Kissmann. Exploiting the computational power of the graphics card: Optimal state space planning on the GPU. In *ICAPS*. AAAI, 2011.
604. M. G. Summa, L. Bottou, B. Goldfarb, F. Murtagh, C. Pardoux, and M. Touati, editors. *Statistical Learning and Data Science*. CRC Computer Science & Data Analysis. Chapman & Hall, 2011.
605. R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
606. Symantec. Symantec internet security threat report - trends for 2010, 2011.
607. M. Tak, M. Winands, and Y. Björnsson. N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012.
608. M. Tambe. *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*. Cambridge University Press, 2011.
609. P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
610. M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani. A detailed analysis of the KDD CUP 99 data set. In *IEEE Intern. Conference on Computational Intelligence for Security and Defense Applications (CISDA)*, pages 53–58, 2009.
611. M. Thielscher. A general game description language for incomplete information games. In M. Fox and D. Poole, editors, *24th AAAI Conference on Artificial Intelligence (AAAI)*, pages 994–999. AAAI Press, 2010.
612. W. M. Thorburn. The myth of Occam's razor. *Mind*, 27(107):345–353, 1918.
613. M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.
614. J. Tiihonen and A. Felfernig. Towards recommending configurable offerings. *International Journal of Mass Customisation*, 3(4):389–406, 2010.
615. P. Torasso and G. Torta. Computing minimal-cardinality diagnoses using BDDs. In *KI*, pages 224–238, 2003.
616. Á. Torralba and V. Alcázar. Constrained symbolic search: On mutexes, BDD minimization and more. In *SOCS*, 2013.
617. A. Torralba, V. Alcázar, D. Borrajo, P. Kissmann, and S. Edelkamp. Symba*: A symbolic bidirectional a* planner. In *International Planning Competition*, pages 105–108, 2014.
618. Á. Torralba, V. Alcázar, P. Kissmann, and S. Edelkamp. Efficient symbolic search for cost-optimal planning. *Artif. Intell.*, 242:52–79, 2017.
619. Á. Torralba, S. Edelkamp, and P. Kissmann. Transition trees for cost-optimal symbolic planning. In *ICAPS*, 2013.
620. Á. Torralba, C. Linares López, and D. Borrajo. Symbolic merge-and-shrink for cost-optimal planning. In *IJCAI*, 2013.
621. C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994 – 12000, 2009.
622. Y.-H. H. Tsai, S. Bai, M. Yamada, L.-P. Morency, and R. Salakhutdinov. Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. *arXiv preprint arXiv:1908.11775*, 2019.
623. I. Tsang, A. Kocsor, and J. T. Kwok. Simpler core vector machines with enclosing balls. In *ICML*, pages 911–918. ACM, 2007.
624. H.-E. Tseng, C.-C. Chang, and S.-H. Chang. Applying case-based reasoning for product configuration in mass customization environments. *Expert Systems with Applications*, 29(4):913–925, 2005.
625. J. W. H. M. Uiterwijk, H. J. van den Herik, and L. V. Allis. A knowledge-based approach to connect four: The game is over, white to move wins. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pages 113–133. Ellis Horwood, 1989.

626. J. Undercoffer, J. Pinkston, A. Joshi, and T. Finin. A target-centric ontology for intrusion detection. In *IJCAI-03 Workshop on Ontologies and Distributed Systems*, pages 47–58, 2004.
627. M. Vallati, L. Chrpa, and T. L. McCluskey. What you always wanted to know about the deterministic part of the international planning competition (IPC) 2014 (but were too afraid to ask). *Knowledge Eng. Review*, 33:e3, 2018.
628. A. Valmari. A stubborn attack on state explosion. *LNCS*, 531:156–165, 1991.
629. D. Van der Weken, S. Van Assche, D. Clabaut, S. Desmet, and B. Volckaert. Automating workflows with service oriented media applications. In *Services - I, 2009 World Conference on*, pages 507–514, 2009.
630. V. N. Vapnik and A. Y. Chervonenkis. *Theory of Pattern Recognition [in Russian]*. Nauka, USSR, 1974.
631. M. Y. Vardi. Boolean satisfiability: Theory and engineering. *Commun. ACM*, 57(3):5, 2014.
632. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30:5998–6008, 2017.
633. K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2009.
634. O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, and D. Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575, 11 2019.
635. J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.
636. Y. Vorobeychik and M. Pritchard. Plan interdiction games. In *Adaptive Autonomous Secure Cyber Systems*, pages 159–182. Springer, 2020.
637. M. vos Savant. Ask Marilyn. *Parade Magazine*, 1990.
638. D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA*, pages 776–787, 2003.
639. K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl. Constrained k-means clustering with background knowledge. In *ICML*, pages 577–584, Williams College, Williamstown MA, 2001.
640. L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.
641. T. T. Wang, A. Gleave, N. Belrose, T. Tseng, J. Miller, M. D. Dennis, Y. Duan, V. Pogrebniak, S. Levine, and S. Russell. Adversarial policies beat professional-level go ais. *CORR*, 2211.00241, 2022.
642. X. Wang, Y. Wu, and D. Zhu. A new variant of in-place sort algorithm. *Procedia Eng.*, 29:2274–2278, 2012.
643. Y. Wang, Z. R. Shi, L. Yu, Y. Wu, R. Singh, L. Joppa, and F. Fang. Deep reinforcement learning for green security games with real-time information. In *AAAI Conference on Artificial Intelligence*, 2019.
644. T. Warden, R. Porzel, J. D. Gehrke, O. Herzog, H. Langer, and R. Malaka. Towards Ontology-based Multiagent Simulations: The PlaSMA Approach. In *European Conference on Modelling and Simulation (ECMS)*, pages 50–56, 2010.
645. Y. Warsame, S. Edelkamp, and E. Plaku. Energy-aware multi-goal motion planning guided by monte carlo search. In *CASE*, pages 335–342. IEEE, 2020.
646. D. F. Watson. Computing the n-dimensional delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2), 1981.
647. B. Webb. Singular value decomposition for collaborative filtering, 2006.
648. I. Wegener. The worst case complexity of McDiarmid and Reed's variant of Bottom-Up Heapsort is less than $n\log n + 1.1n$. *Information and Computation*, 97(1):86–96, 1992.
649. I. Wegener. Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.
650. I. Wegener. *Branching Programs and Decision Diagrams*. SIAM, 2000.
651. I. Wegener. *Komplexitätstheorie*. Springer, 2003. (in German).
652. G. M. Weiss and H. Hirsh. Learning to predict rare events in categorical time-series data. In *Proceedings of the 1998 AAAI/ICML Workshop on Time-Series Analysis, Madison, Wisconsin*, 1998.
653. J. P. Wergin. *Wergin on Skat and Sheepshead*. Wergin Distributing, Mc. Farland, USA, 1975.
654. A. Wijs. *What to do Next? Analysing and Optimising System Behaviour in Time*. PhD thesis, Vrije Universiteit Amsterdam, 1999.
655. S. Wild and M. E. Nebel. Average case analysis of java 7's dual pivot quicksort. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 825–836, 2012.
656. S. Wild, M. E. Nebel, and R. Neininger. Average case and distributional analysis of dual-pivot quicksort. *ACM Transactions on Algorithms*, 11(3):22:1–22:42, 2015.
657. J. W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–348, 1964.
658. M. H. Winands, Y. Björnsson, and J.-T. Saito. Monte-carlo tree search solver. *Computers and Games*, 5131:25–36, 2008.
659. Q. Wu and J.-K. Hao. Coloring large graphs based on independent set extraction. *Computers and Operations Research*, 39:283–290, 2012.
660. S. Wu, M. Kay, R. King, A. Vila-Parrish, and D. Warsing. Multi-objective optimization of 3D packing problem in additive manufacturing. In *Industrial and Systems Engineering Research Conference*, 2014.
661. X. Xing, T. Warden, T. Nicolai, and O. Herzog. SMIZE: A Spontaneous Ride-Sharing System for Individual Urban Transit. In *German Conference on Multiagent System Technologies. (MATES)*, pages 165–176, Berlin, 2009. Springer.
662. G. Xunrang and Z. Yuzhang. A new Heapsort algorithm and the analysis of its complexity. *Comput. J.*, 33(3):281–282, 1990.
663. G. Xunrang and Z. Yuzhang. Asymptotic optimal Heapsort algorithm. *Theoret. Comput. Sci.*, 134(2):559–565, 1994.
664. G. Xunrang and Z. Yuzhang. Optimal heapsort algorithm. *Theoret. Comput. Sci.*, 163(1–2):239–243, 1996.

665. X. Yan, P. Diaconis, P. Rusmevichientong, and B. V. Roy. Solitaire: Man versus machine. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1553–1560. MIT Press, Cambridge, MA, 2004.
666. V. Yaroslavskiy. Dual-Pivot Quicksort algorithm, 2009.
667. D. Yu and D. Frincke. Improving the quality of alerts and predicting intruder's next goal with hidden colored petri-net. *Comput. Netw.*, 51:632–654, February 2007.
668. W. Yu, M. Li, and X. Li. Optimizing pyramid visibility coverage for autonomous robots in 3D environment. *Control and Intelligent System*, 42:9–15, 2014.
669. D. Zastrau and S. Edelkamp. Stochastic gradient descent with GPGPU. In *KI*, volume 7526 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2012.
670. W. Zhang. Truncated branch-and-bound: A case study on the asymmetric TSP. In *AAAI-93 Spring Symposium on AI and NP-Hard Problems*, pages 160–166, 1993.
671. W. Zhang. Depth-first branch-and-bound versus local search: A case study. In *Proceedings of the National Conference on Artificial Intelligence*, pages 930–936, 2000.
672. Y.-F. Zhang, Z.-Y. Xiong, and X.-Q. Wang. Distributed intrusion detection based on clustering. In *Proceedings of the Fourth International Conference on Machine Learning and Cybernetics*, 2005.
673. Z. Zhang, S. Guo, W. Zhu, W.-C. Oon, and A. Lim. Space defragmentation heuristic for 2D and 3D bin packing problems. In *IJCAI*, pages 699–704, 2011.
674. R. Zhou and E. Hansen. Sweep A*: space-efficient heuristic search in partially ordered graphs. In *15th IEEE International Conference on Tools with Artificial Intelligence*, pages 427–434, 2003.
675. R. Zhou and E. A. Hansen. Multiple sequence alignment using A*. In *AAAI*, 2002. Student abstract.
676. R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–689, 2004.
677. R. Zhou and E. A. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.
678. M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pages 1729–1736, 2008.