Habilitationsschrift
zur Erlangung der Venia legendi der
Fakultät für angewandte Wissenschaften
der Albert-Ludwigs-Universität Freiburg
für das Fach Informatik

# Heuristic Search

Stefan Edelkamp

September 19, 2002

# Summary

Representing and exploring large and even infinite state spaces is a challenge for many subareas of computer science in general and for Artificial Intelligence (AI) in particular. A possibly weighted state space graph is to be explored by successively applying state expansions starting from the initial state to eventually reach a desired goal state.

This thesis studies the theoretical foundations of guided exploration algorithms and their impact in the following application areas.

**Puzzle Solving** Domain-dependent puzzle solving searches for optimal solution paths in solitaire games. Challenges are real-world problems like Rubik's Cube or the set of $(n^2 - 1)$-Puzzles, and computer games like Sokoban or Atomix. The thesis takes Atomix as a selected case study for which it refines storage structures to enhance duplicate detection.

**Action Planning** In action planning, domains and problem instances are specified in a general description language (PDDL) with parameterized operators. Starting from propositional planning, in which each state is represented as a subset of atoms, planning problems have been extended with action duration, resource variables, and objective functions. The thesis proposes a successful metric and temporal heuristic search framework planner featuring a versatile static analyzer and different heuristic estimates.

**Theorem Proving** Automated deduction considers object logics like first and higher-order logic to specify axioms and theorems, searching for according proofs. Many current theorem provers face infinite spaces of all possible proof states. The thesis proposes functional heuristic search to generate proofs fully automatically.

**Hardware Verification** Hardware designs may contain subtle errors. These are to be found with automated verification techniques that validate if an implemented system is conformant to its specification. The thesis adds symbolic heuristic search to the algorithmic portfolio of an existing $\mu$-calculus model checker based on an estimate that propagates the error description through the circuit.

**Software Verification** Asynchronous software systems like communication protocols or multi-threaded Java programs require involved concurrency maintenance e.g. to avoid deadlocks. Automated validation simulates all possible executions traces and yields short witnesses in case of a failure. This thesis contributes explicit-state directed model checking and shows how the control of flow can be inferred by supervised learning by example.

**Route planning** Traffic information systems search for lowest-cost paths in explicit maps. If the map is provided on CD/DVD, it is likely too large to be kept in main memory. The thesis proposes a complete and optimal localized heuristic search exploration scheme that explicitly pages portions of the graph according to the spatial structure of the map. It further considers route planning and map inference aspects for a set of global positioning traces.

The central problem in all research areas is overcoming the state (space) explosion problem: many puzzles are known to be at least NP-hard, propositional planning is

PSPACE-complete, while numeric planning and automated theorem proving are both undecidable. Last but not least, the state space size in concurrent systems grows often exponentially in the number of state variables.

To cope with this intrinsic hardness, all contributed algorithms apply heuristic search to focus the search process, where the estimates are found by exploring abstracted state spaces.

Among other techniques to reduce the number of nodes to be looked at, the thesis studies state compaction, symbolic representation, pattern databases, and partial order reduction. State compaction stores a small signature of the state, symbolic representation assigns and maintains characteristic formulae for sets of states, pre-computed dictionaries serve refined estimates for the overall search engines, and partial order reduction exploits the commutativity of concurrent actions.

Beside theoretical work we provide implementations of heuristic search algorithms in practical systems, including a commercial car navigation system, an awarded action planner, an automated higher-order theorem prover, a generic puzzle solver, a programming-by-example tool, a symbolic and an explicit-state model checker.

## Acknowledgments

Thanks to

- Prof. Dr. Richard Korf and Dr. Ulrich Meyer for their cooperation in *Foundations*
- Falk Hueffner for his cooperation in *Puzzle Solving*
- Dr. Frank Reffel for his cooperation in *Hardware Verification*
- Alberto Lluch-Lafuente and Prof. Dr. Stefan Leue for their cooperation in *Software Verification*
- Malte Helmert for his cooperation in *Action Planning*
- Dr. Stefan Schroedl for his cooperation in *Route Planning*
- Peter Leven for his cooperation in *Theorem Proving*
- Prof. Dr. Thomas Ottmann for his overall support

## Dedication

To Gisela and Monika, *zwei Mädchen aus Germany* ...

*Freiburg, September 19, 2002* Stefan Edelkamp

# Overview

The thesis divides into seven parts: *Foundations*, *Puzzle Solving*, *Action Planning*, *Theorem Proving*, *Software Verification*, *Hardware Verification*, and *Route Planning*. It is composed of a selection published papers, organized in form of chapters. The order of the presentation has been chosen to maximize comprehensiveness.

## Foundations

For Part I we selected the following four papers.

- Memory Limitation in Artificial Intelligence.
- Theory and Practice of Time-Space Trade-Offs in Memory Limited Search.
- Time Complexity of Iterative-Deepening-A*.
- Prediction of Regular Search Tree Growth by Spectral Analysis.

The first paper surveys approaches to cope with limited memory for the design of algorithms and data structures in Artificial Intelligence, since many systems, for instance in puzzle solving, two-player games, action and route planning, robotics, machine learning, data mining, logic programming, and theorem proving, explore very large or even infinite implicitly given state spaces. The main focus is on strategies for an improved time-space trade-off in case main memory becomes exhausted and include refined representation and exploration techniques, the introduction of additional search or control knowledge, and the explicit maintenance of secondary memory. The paper is utilized as a welcome introduction to the topic of the thesis.

The second paper presents theoretical and practical results on new variants for exploring state-space with respect to memory limitations. The work refines storage maintenence aspects from the first one. It establishes $O(\log n)$ minimum-space algorithms that omit both the open and the closed list to determine the shortest path between every two nodes and studies the gap in between full memorization in a hash table and the information-theoretic lower bound. The proposed structure of suffix-lists elaborates on a concise binary representation of states by applying bit-state hashing techniques. Significantly more states can be stored while searching and inserting $n$ items into suffix lists is still available in $O(n \log n)$ time. Bit-state hashing leads to the new paradigm of partial iterative-deepening heuristic search, in which full exploration is sacrificed for a better detection of duplicates in large search depth.

The third paper analyzes the time complexity of IDA*. We first show how to calculate the exact number of nodes at a given depth of a regular search tree, and the asymptotic brute-force branching factor. We then use this result to analyze IDA* with a consistent, admissible heuristic function. Previous analyses relied on an abstract analytic model, and characterized the heuristic function in terms of its accuracy, but do not apply to concrete problems. In contrast, our analysis allows us to accurately predict the performance of IDA* on actual problems such as the sliding-tile puzzles and Rubik's Cube. The heuristic function is characterized by the distribution of heuristic values over the problem space. Contrary to conventional wisdom, our analysis shows that the asymptotic

heuristic branching factor is the same as the brute-force branching factor. Thus, the effect of a heuristic function is to reduce the effective depth of search by a constant, relative to a brute-force search, rather than reducing the effective branching factor.

The last paper elaborates on the above result, which shows that predicting the growth of the search tree in IDA* essentially relies on only two criteria: The number of nodes in the brute-force search tree for a given depth and the equilibrium distribution of the heuristic estimate. Since the latter can be approximated by random sampling, the paper accurately predicts the number of nodes in the brute-force search tree for large depth in closed form by analyzing the spectrum of the problem graph or one of its factorization. It further derives that the asymptotic brute-force branching factor is in fact the spectral radius of the problem graph and exemplifies the considerations in the domain of the $(n^2 - 1)$-Puzzle.

## Puzzle Solving

In Part II we chose the paper that documents current aspects to solve challenges in solitaire games best.

- Finding Optimal Solutions to Atomix

We present solutions of benchmark instances to the solitaire computer game Atomix found with different heuristic search methods. The problem is PSPACE-complete. An implementation of the heuristic algorithm A* is presented that needs no priority queue, thereby having very low memory overhead. The limited memory algorithm IDA* is handicapped by the fact that, due to move transpositions, duplicates appear very frequently in the problem space; several schemes of using memory to mitigate this weakness are explored, among those, “partial” schemes which trade memory savings for a small probability of not finding an optimal solution. Even though the underlying search graph is directed, backward search is shown to be viable, since the branching factor can be proven to be the same as for forward search.

## Action Planning

In Part III we present a total of five papers that document the basics and novelties in the development of our planning system MIPS.

- Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length.
- The Model Checking Integrated Planning System.
- Directed Symbolic Exploration and its Application to AI-Planning.
- Planning with Pattern Databases.
- Symbolic Pattern Databases in Heuristic Search Planning.
- Taming Numbers and Durations in the Model Checking Integrated Planning System.

In the first paper we present a general-purposed algorithm for transforming a planning problem specified in STRIPS into a concise state description for single state or symbolic exploration. The process of finding a state description consists of four phases. In Phase 1 we symbolically analyze the domain specification to determine constant and one-way predicates, i.e. predicates that remain unchanged by all operators or toggle in only one direction, respectively. In Phase 2 we symbolically merge predicates which leads to a drastic reduction of state encoding size, while in Phase 3 we constrain the domains of the predicates to be considered by enumerating the operators of the planning problem. Phase 4 combines the result of the previous phases.

The second paper describes the system at 2000 as a search engine that applies binary decision diagrams to compactly represent world states in a planning problem and efficiently explore the underlying state space. It is the first general planning system based on model checking methods. It can handle the STRIPS subset of the PDDL language and some additional features from ADL, namely negative preconditions and (universal) conditional effects. At the AIPS 2000 conference, MIPS has been one of five planning systems to be awarded for "Distinguished Performance" in the fully automated track. The article gives a brief introduction to BDDs and explains the basic planning algorithm employed by MIPS, using a simple logistics problem as an example.

The third paper studies traditional and enhanced BDD-based exploration procedures capable of handling large planning problems. As shown above, reachability analysis and model checking have eventually approached AI-Planning. Unfortunately, they (typically) rely on uninformed *blind* search. On the other hand, heuristic search and especially lower bound techniques have matured in effectively directing the exploration even for large problem spaces. Therefore, with heuristic symbolic search we address the unexplored middle ground between single state and symbolic planning engines to establish algorithms that can gain from both sides. To this end we implement and evaluate heuristics found in state-of-the-art heuristic single-state search planners.

The forth and fifth paper contribute a new heuristic, since previous estimates were either not admissible or too weak, so that optimal solutions were found in rare cases only. In contrast, heuristic pattern databases are known to significantly improve lower-bound estimates for optimally solving challenging single-agent problems like the 24-Puzzle and Rubik's Cube. The paper studies the effect of pattern databases in the context of deterministic planning. Given a fixed state description based on instantiated predicates, we provide a general abstraction scheme to automatically create admissible domain-independent memory-based heuristics for planning problems, where abstractions are found in factorizing the planning space. We evaluate the impact of pattern database heuristics in A* and hill climbing algorithms for a collection of benchmark domains.

Symbolic pattern databases (SPDB) combine the two aspects heuristic search and model checking, being off-line computed dictionaries, that are generated in symbolic backward traversals of automatically inferred planning space abstractions. The entries of SPDBs serve as heuristic estimates to accelerate explicit and symbolic, approximate and optimal heuristic search planners. Selected experiments highlight that the symbolic representation yields much larger and more accurate pattern databases than the ones generated with explicit methods.

Since the MIPS has also shown distinguished performance in the third international planning competition the sixth paper present the status quo of the object-oriented framework architecture that clearly separates the portfolio of explicit and symbolic heuristic

search exploration algorithms from different on-line and off-line computed estimates and from the planning problem representation. The planner extensions include critical path analysis of sequentially generated plans to generate optimal parallel plans. The linear time algorithm bypasses known NP hardness results for partial ordering with mutual exclusion by scheduling plans with respect to the set of actions *and* the imposed causal structure. To improve exploration guidance approximate plans are scheduled for each encountered planning state. One major strength of MIPS is its static analysis phase that grounds and simplifies parameterized predicates, functions and operators, that infers single-valued invariances to minimize the state description length, and that detects symmetries of domain objects. The aspect of object symmetry is analyzed in detail. The paper shows how temporal plans of any planner can be visualized in Gannt-chart format in a client-server architecture. The front-end turns also be appropriate for concise almost problem instance independent domain visualization.

## Theorem Proving

Part IV provides one paper which presents recent work, that analyzes the effect of heuristic search algorithms like A* and IDA* to accelerate proof-state based theorem provers.

- Directed Automated Theorem Proving.

A functional implementation of possibly weighted A* is proposed that extends Dijkstra's single-source shortest-path algorithm. Efficient implementation issues and possible flaws for both A* and IDA* are discussed in detail.

Initial results with first and higher order logic examples in *Isabelle* indicate that *directed automated theorem proving* is superior to other known general inference mechanisms and that it can enhance other proof techniques like model elimination.

## Software Verification

In Part V, the correctness of software mostly in form of communication protocols is considered. The selection consists of four papers.

- Inferring Flow of Control in Program Synthesis by Example.
- Directed Explicit-State Model Checking in the Validation of Communication Protocols.
- Trail-Directed Model Checking.
- Partial-Order Reduction in Directed Model Checking.

The first paper presents a supervised, interactive learning technique that infers control structures of computer programs from user-demonstrated traces. A two-stage process is applied: first, a minimal deterministic finite automaton (DFA) $M$ labeled by the instructions of the program is learned from a set of example traces and membership queries to the user. It accepts all prefixes of traces of the target program. The number of queries

is bounded by $O(k \cdot |M|)$, with $k$ being the total number of instructions in the initial example traces. In the second step we parse this automaton into a high-level programming language in $O(|M|^2)$ steps, replacing jumps by conditional control structures.

The next three papers are different from the former approach in the sense that they address the verification instead of the synthesis problem, which automation is known as model checking. The success of model checking is largely based on its ability to efficiently locate errors in software designs. If an error is found, a model checker produces a trail that shows how the error state can be reached, which greatly facilitates debugging. However, while current model checkers find error states efficiently, the counterexamples are often unnecessarily lengthy, which hampers error explanation. This is due to the use of "naive" search algorithms in the state space exploration.

In the second paper we present approaches to the use of heuristic search algorithms in explicit-state model checking. We present the class of A* directed search algorithms and propose heuristics together with bit-state compression techniques for the search of safety property violations. We achieve great reductions in the length of the error trails, and in some instances render problems analyzable by exploring a much smaller number of states than standard depth-first search. We then suggest an improvement of the nested depth-first search algorithm and show how it can be used together with A* to improve the search for liveness property violations. Our approach to directed explicit-state model checking has been implemented in a tool set called HSF-SPIN. We provide experimental results from the protocol validation domain using HSF-SPIN.

The second paper improves HSF-SPIN as a Promela model checker based on heuristic search strategies. It utilizes heuristic estimates in order to direct the search for finding software bugs in concurrent systems. As a consequence, HSF-SPIN is able to find shorter trails than blind depth-first search. The paper contributes an extension to the paradigm of *directed model checking* to shorten already established unacceptable long error trails. This approach has been implemented in HSF-SPIN. For selected benchmark and industrial communication protocols experimental evidence is given that *trail-directed model checking* effectively shortcuts existing witness paths.

The third paper presents one refinement to the one above by partial order reduction as a very succesful technique for avoiding the state explosion problem that is inherent to explicit state model checking of asynchronous concurrent systems. It exploits the commutativity of concurrently executed transitions in interleaved system runs in order to reduce the size of the explored state space. Directed model checking on the other hand addresses the state explosion problem by using guided search techniques during state space exploration. As a consequence, shorter errors trails are found and less search effort is required than when using standard depth-first or breadth-first search. We analyze how to combine directed model checking with partial order reduction methods and give experimental results on how the combination of both techniques performs.

## Hardware Verification

Part VI shows how to apply symbolic heuristic search to find errors in hardware circuits. Actually this was the first application area we applied the technique to.

- Error Detection with Directed Symbolic Model Checking.

In practice due to entailed memory limitations the most important problem in model checking is state (space) explosion. Therefore, to prove the correctness of a given design, binary decision diagrams (*BDDs*) are widely used as a concise and symbolic state space representation. Nevertheless, *BDDs* are not able to avoid an exponential blow-up in general. If we restrict ourselves to find an error of a design which violates a safety property, in many cases a complete state space exploration is not necessary and the introduction of a heuristic to guide the search can help to keep both the explored part and the associated *BDD* representation smaller than with the classical approach. In the paper we will show that this idea can be extended with an automatically generated heuristic and that it is applicable to a large class of designs. Since the proposed algorithm can be expressed in terms of *BDDs* it is even possible to use an existent model checker without any internal changes.

## Route Planning

Part VII addresses the problem of finding shortest paths in large external or dynamic maps and consists of the following two papers.

- Localizing A*.
- Route Planning and Map Inference with Global Positioning Traces.

The first paper acknowledges the fact that heuristic search in large problem spaces inherently calls for algorithms capable of running under restricted memory. This question has been investigated in a number of articles. However, in general the efficient usage of two-layered storage systems is not further discussed. Even if hard-disk capacity is sufficient for the problem instance at hand, the limitation of *main memory* may still represent the bottleneck for their practical applications. Since breadth-first and best-first strategies do not exhibit any locality of expansion, standard *virtual memory management* can soon result in thrashing due to excessive page faults. In the paper we propose a new search algorithm and suitable data structures in order to minimize page faults by a local reordering of the sequence of expansions. We prove its correctness and completeness and evaluate it in a real-world scenario of searching a large road map in a commercial route planning system.

Navigation systems assist almost any kind of motion in the physical world including sailing, flying, hiking, driving and cycling. On the other hand, traces supplied by global positioning systems (GPS) can track actual time and absolute coordinates of the moving objects. Consequently, this last paper addresses efficient algorithms and data structures for the route planning problem based on GPS data; given a set of traces and a current location, infer a short(est) path to the destination. The algorithm of Bentley and Ottmann is shown to transform geometric GPS information directly into a combinatorial weighted and directed graph structure, which in turn can be queried by applying classical and refined graph traversal algorithms like Dijkstras' single-source shortest path algorithm or A*. For high-precision map inference especially in car navigation, algorithms for road segmentation, map matching and lane clustering are presented.

# Omissions and Demarcation

For the sake of conciseness and integrity of this thesis the work does not include all available publications. In the following we briefly mention the papers that have been left out for various reasons.

In *Foundations* the following work on two-player games has not been included, since it does not refer to *heuristic search*.

- Stefan Edelkamp. Symbolic Exploration in Two-Player Games: Preliminary Results. In Artificial Intelligence Planning and Scheduling (AIPS)–Workshop on Model Checking, 2002.

In this paper symbolic exploration with binary decision diagrams (BDDs) is applied to two-player games to improve main memory consumption for reachability analysis and game-theoretical classification, since BDDs provide a compact representation for large set of game positions. A number of examples are evaluated: *Tic-Tac-Toe*, *Nim*, *Hex*, and *Four Connect*. In *Chess* we restrict the considerations to the creation of endgame databases. The results are prelimary, but the study puts forth the idea that BDDs are widely applicable in game playing and provide a universal tool for people interested in quickly solving practical problems.

In the part *Action Planning* we left out the following initial paper to retain a concise representation in favor to more elaborated results.

- Stefan Edelkamp and Frank Reffel. Deterministic State Space Planning with BDDs. In *European Conference on Planning (ECP)*, pages 381–382, Lecture Notes in Computer Science (Preprint), Springer, 1999.

The short paper (for a long version cf. [113]) proposes a planner that applies BDDs to compactly represent sets of propositionally represented states. Using this representation, accurate reachability analysis and backward chaining can apparently be carried out without necessarily encountering exponential representation explosion. The main objectives are the interest in optimal solutions, the generality and the conciseness of the approach. The algorithms are tested against the AIPS'98 planning competition problems and lead to substantial improvements to existing solutions.

We further omitted two papers from this part, which are subsumed by a subsequent journal publication.

- Stefan Edelkamp. First Solutions to PDDL+ Planning Problems. In *Artificial Intelligence Planning and Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*, 75-88, 2001.
- Stefan Edelkamp. Mixed Propositional and Numerical Planning in the Model Checking Integrated Planning System Preliminary Results. In *Artificial Intelligence Planning and Scheduling (AIPS)–Workshop on Temporal Planning*, 2002.

In the first paper we present the design and algorithmic details of a directed search temporal and metric planner to solve benchmark planning problems specified in PDDL+

syntax. The planner produces sequential solutions and handles mixed logical-numerical problems, grounds and groups predicates, simplifies arithmetic trees, and instantiates numerical quantities on the fly. It uses forward state-space search with an A* search control and is evaluated in problem instances of two benchmark planning problems.

The second paper considers the extensions to the planner to solve mixed propositional, temporal and numerical planning problems in PDDL+ syntax for the 3rd international planning competition. The directed search exploration algorithm applies critical path scheduling to parallelize sequential plans.

In *Software Verfication* two conference papers were omitted, since they have also been merged and extended in a journal publication.

- Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

- Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed Model-Checking in HSF-SPIN. In *SPIN Workshop*, pages 57–79, Lecture Notes in Computer Science. Springer, 2001.

In the first paper we present an approach to reconcile explicit state model checking and heuristic directed search suited to an AI audience. We provide experimental evidence that the model checking problem for concurrent systems, such as communications protocols, can be solved more efficiently, since finding a state violating a property can be understood as a directed search problem.

The second paper addresses the model checking audience and also incorporates a large class of LTL-specified liveness properties. We propose an improved nested depth-first search algorithm that exploits the structure of Promela Never-Claims and provide experimental results for applying HSF-SPIN to two toy protocols and one real world protocol, the CORBA GIOP protocol.

The following contributions were excluded from the manuscript, since they are already cited and discussed in the Ph.D. thesis: *Data Structures and Learning Algorithms in State Space Search (Datenstrukturen und Lernverfahren in der Zustandsraumsuche)*, DISKI-201, Infix, 1999.

- Stefan Edelkamp and Jürgen Eckerle. New Strategies In Real-Time Heuristic Search. In *National Conference on Artificial Intelligence (AAAI)-Workshop on Online Search*, pages 30–35, 1997.

- Stefan Edelkamp and Stefan Schrödl. Learning Dead Ends in Sokoban In *Workshop Komplexitätstheorie, Datenstrukturen und Effiziente Algorithmen*, Technical Report CSR-98-01, Fakultät für Informatik, TU Chemnitz, pages 16–21, 1998.

- Stefan Edelkamp. GST: General Sliding Tile In *Workshop Komplexitätstheorie, Datenstrukturen und Effiziente Algorithmen*, Technical Report, Institut für Informatik, Universität Mainz, 1997

- Stefan Edelkamp and Richard E. Korf. The Branching Factor of Regular Search Spaces. In *National Conference on Artificial Intelligence (AAAI)*, 1998. 299–304.

- Stefan Edelkamp. Suffix Tree Automata in State Space Search. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science, pages 381–385. Springer, 1997.

- Stefan Edelkamp. Dictionary Automaton in Optimal Space, 1999. Technical Report 129, Institut für Informatik, Albert-Ludwigs-Universität Freiburg.

- Stefan Edelkamp. Updating Shortest Paths. In *European Conference on Artificial Intelligence (ECAI)*, pages 655–659. Wiley, 1998.

- Stefan Edelkamp and Frank Reffel. OBDDs in Heuristic Search. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science, pages 81–92. Springer, 1998.

The next two papers were published after the Ph.D. was accomplished, but were omitted, since they are written in German.

- Stefan Edelkamp. Datenstrukturen und Lernverfahren in der Zustandsraumsuche. KI, 3, pages 49-51, ArenDTaP, 1999.

- Stefan Edelkamp. Neue Wege in der Exploration. In *Informatik*, Lecture Notes in Computer Science, pages 65–77. Springer, 2000.

Furthermore, the following work was excluded, since it refers to a dictionary or a multi-media product.

- Phillip A. Laplante, editor. *Dictionary of Computer Science, Engeneering and Technology*. CRC Press, 2000. One contributor: Stefan Edelkamp.

- Thomas Ottmann, Sven Schuierer and Stefan Edelkamp. *Geometrische Algorithmen*. VIROR - AOF Compact Disc, 1999. http://www.viror.de/lernen/cdroms.

- Thomas Ottmann, Alois Heinz and Stefan Edelkamp. *Algorithmentheorie*. VIROR - AOF Compact Disc, 2000. http://www.viror.de/lernen/cdroms.

Last but not least, two recent papers on sequential sorting were not mentioned in the main part of this thesis, since they do not fit well to the topic of state-space search.

- Stefan Edelkamp and Ingo Wegener. On the Performance of Weak-Heapsort In *Symposium on Theoretical Aspects in Computer Science Computer Science (STACS)*, Lecture Notes in Computer Science, pages 254–266. Springer, 2000. (An extended version is published in Electronic Colloquium on Computational Complexity, TR99-028, ISSN 1433-8092.)

- Stefan Edelkamp and Patrick Stiegeler. Pushing the Limits in Sequential Sorting. In *Workshop on Algorithm Engineering (WAE)*, Lecture Notes in Computer Science. Springer, 2000. (A fairly extended version is going to be published in the *ACM Journal of Experimental Algorithmics* with the title *Implementing HEAPSORT with* $n \log n - 0.9n$ *and QUICKSORT with* $n \log n + 0.2n$ *comparisons*).

The former paper presents a further *HEAPSORT* variant called *WEAK-HEAPSORT*, which also contains a new data structure for priority queues. The sorting algorithm and the underlying data structure are analyzed showing that *WEAK-HEAPSORT* is the best *HEAPSORT* variant and that it has a lot of nice properties. It is shown that the worst case number of comparisons is $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + n - \lceil\log n\rceil \leq n\log n + 0.1n$ and *Weak-Heaps* can be generated with $n-1$ comparisons. A double-ended priority queue based on *Weak-Heaps* can be generated in $n + \lceil n/2\rceil - 2$ comparisons. Moreover, examples for the worst and the best case of *WEAK-HEAPSORT* are presented, the number of *Weak-Heaps* on $\{1,\ldots,n\}$ is determined, and experiments on the average case are reported.

The latter paper presents refinements to the *WEAK-HEAPSORT* algorithm establishing the general and practical relevant sequential sorting algorithm *INDEX-WEAK-HEAPSORT* with exactly $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + 1 \leq n\log n - 0.9n$ comparisons and at most $n\log n + 0.1n$ transpositions on any given input. It comprises an integer array of size $n$ and is best used to generate an index for the data set. With *RELAXED-WEAK-HEAPSORT* and *GREEDY RELAXED-WEAK-HEAPSORT* we discuss modifications for a smaller set of pending index element transpositions. If extra space to create an index is not available, with *QUICK-WEAK-HEAPSORT* we propose an efficient *QUICKSORT* variant with $n\log n + 0.2n + o(n)$ comparisons on the average. Furthermore, we present data showing that *WEAK-HEAPSORT*, *INDEX-WEAK-HEAPSORT* and *QUICK-WEAK-HEAPSORT* beat other performant *QUICKSORT* and *HEAPSORT* variants even for moderate values of $n$.

# Contents


## I Foundations 1

**1 Memory Limitation in Artificial Intelligence 3**
1.1 Introduction . . . 4
1.2 Hierarchical Memory . . . 4
1.3 Single-Agent Search . . . 6
1.4 Action Planning . . . 10
1.5 Game Playing . . . 14
1.6 Other AI Areas . . . 16
1.7 Conclusions . . . 17

**2 Time-Space Trade-Offs in Memory Limited Search 19**
2.1 Introduction . . . 20
2.2 Minimum Space Algorithms . . . 21
2.2.1 Divide-And-Conquer BFS . . . 21
2.2.2 Divide-And-Conquer SSSP . . . 22
2.3 Suffix Lists . . . 23
2.3.1 Representation . . . 23
2.3.2 Searching . . . 24
2.3.3 Inserting . . . 24
2.3.4 Checkpoints . . . 24
2.3.5 Buffers . . . 24
2.3.6 The Information Theoretic Bound . . . 26
2.4 Bit-State Hash-Tables . . . 27
2.4.1 State Space Search for Protocols Validation . . . 27
2.4.2 Supertrace . . . 27
2.4.3 Data Structures . . . 28
2.4.4 Sequential and Universal Hashing . . . 28
2.4.5 Validating Process . . . 30
2.4.6 Heuristic Search Algorithm . . . 30
2.5 Conclusion . . . 31

**3 Time Complexity of Iterative-Deepening-A* 35**
3.1 Introduction and overview . . . 36
3.2 Branching factor of regular search trees . . . 36
3.2.1 Graph versus tree-structured problem spaces . . . 36
3.2.2 Example: Rubik's Cube . . . 37
3.2.3 A system of simultaneous equations . . . 38

3.2.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.2.5 Generality of this technique . . . . . . . . . . . . . . . . . . . . . 41
3.3 Time complexity of IDA* . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3.1 Previous work . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3.3 Consistent heuristics . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3.4 Conditions for node expansion . . . . . . . . . . . . . . . . . . . 42
3.3.5 Characterization of the heuristic . . . . . . . . . . . . . . . . . . 43
3.3.6 General result . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.3.7 The heuristic branching factor . . . . . . . . . . . . . . . . . . . 47
3.3.8 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . 48
3.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.5 Generality and further work . . . . . . . . . . . . . . . . . . . . . . . . 52

**4 Prediction of Regular Search Tree Growth 55**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.2 Linear Algebra Basics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.3 Partitioning the Search Space . . . . . . . . . . . . . . . . . . . . . . . 58
4.4 Equivalence Graph Structure . . . . . . . . . . . . . . . . . . . . . . . 59
4.5 Exact Prediction of Search Tree Size . . . . . . . . . . . . . . . . . . . 61
4.6 Approximate Prediction of Search Tree Size . . . . . . . . . . . . . . . 63
4.7 Generalizing the Approach . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.7.1 Other Problem Spaces . . . . . . . . . . . . . . . . . . . . . . . 64
4.7.2 Pruning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.7.3 Non-Diagonizability . . . . . . . . . . . . . . . . . . . . . . . . 66
4.7.4 Start Vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.8 Previous Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.9 Conclusion and Discussion . . . . . . . . . . . . . . . . . . . . . . . . . 67

## II Puzzle Solving 69

**5 Finding Optimal Solutions to Atomix 71**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2 Heuristic Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.3 Related Puzzles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.4 Complexity of Atomix . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.4.1 Complexity of Sliding-Block Puzzles . . . . . . . . . . . . . . . . 74
5.4.2 A Formal Definition of Atomix . . . . . . . . . . . . . . . . . . . 75
5.4.3 The Hardness of Atomix . . . . . . . . . . . . . . . . . . . . . . 76
5.5 Searching the State Space of Atomix . . . . . . . . . . . . . . . . . . . 76
5.5.1 Heuristics for Atomix . . . . . . . . . . . . . . . . . . . . . . . . 77
5.5.2 A* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.5.3 IDA* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.5.4 Partial IDA* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.5.5 Backward Search . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.6 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

5.6.1 Identical Atoms . . . 82
5.6.2 A* . . . 82
5.7 Conclusions . . . 83
5.8 Experimental Results . . . 84

**III Action Planning 87**

**6 Exhibiting Knowledge in Planning Problems 89**
6.1 Introduction . . . 90
6.2 Symbolic Exploration . . . 91
6.3 Parsing . . . 92
6.4 Constant and One-Way Predicates . . . 93
6.5 Merging Predicates . . . 94
6.6 Exploring Predicate Space . . . 95
6.6.1 Action-Based Exploration . . . 96
6.6.2 Fact-Based Exploration . . . 97
6.7 Combining Balancing and Exploration . . . 97
6.8 Experimental Results . . . 99
6.9 Related Work and Conclusion . . . 99

**7 The Model Checking Integrated Planning System 103**
7.1 BDDs: Why and For What? . . . 104
7.2 BDDs for Representing Sets of States . . . 105
7.3 BDDs for Representing State Transitions . . . 107
7.4 Evaluation of the MIPS Algorithm . . . 107
7.5 Outlook . . . 109

**8 Directed Symbolic Exploration in AI-Planning 111**
8.1 Introduction . . . 112
8.2 BDD Representation . . . 113
8.3 BDD-Based Blind Search . . . 115
8.3.1 Bidirectional Search . . . 116
8.3.2 Forward Set Simplification . . . 116
8.4 BDD-Based Directed Search . . . 117
8.4.1 Heuristic Pattern Databases . . . 118
8.4.2 BDDA* . . . 121
8.4.3 Best-First-Search . . . 122
8.5 Experiments . . . 123
8.5.1 Gripper . . . 123
8.5.2 Logistics . . . 124
8.5.3 Planning as Model Checking . . . 125
8.6 Conclusion and Outlook . . . 126

**9 Planning with Pattern Databases 129**
9.1 Introduction . . . 130
9.1.1 Optimal Planning Approaches . . . 130
9.1.2 Heuristic Search Planning . . . 130

9.2 Planning Space Representation . . . . . 131
9.3 Pattern Databases . . . . . 132
9.3.1 State Abstractions . . . . . 133
9.3.2 Disjoint Pattern Databases . . . . . 134
9.3.3 Perfect Hashing . . . . . 135
9.3.4 Clustering . . . . . 135
9.4 Results . . . . . 136
9.4.1 Logistics . . . . . 136
9.4.2 Other Domains . . . . . 138
9.5 Conclusion . . . . . 139

**10 Symbolic Pattern Databases 141**
10.1 Introduction . . . . . 142
10.2 Pattern Databases in AI-Planning . . . . . 143
10.2.1 Grounded Propositional Planning . . . . . 143
10.2.2 Abstract Planning problems . . . . . 143
10.2.3 Planning Pattern Databases . . . . . 144
10.2.4 Disjoint Pattern Databases . . . . . 145
10.2.5 Partitions and Storage . . . . . 146
10.3 Symbolic Pattern Databases . . . . . 146
10.3.1 States and Operators . . . . . 146
10.3.2 Pattern Database Construction . . . . . 147
10.4 Explicit Pattern Database Search . . . . . 148
10.5 Symbolic Pattern Database Search . . . . . 149
10.6 Search Tree Prediction . . . . . 150
10.7 Case Study . . . . . 150
10.7.1 Pattern Database Construction . . . . . 151
10.7.2 Explicit Search . . . . . 152
10.7.3 Symbolic Search . . . . . 153
10.8 Related Work . . . . . 155
10.9 Conclusion . . . . . 156

**11 Taming Numbers and Durations in MIPS 159**
11.1 Introduction . . . . . 160
11.2 The Development of MIPS . . . . . 162
11.3 Terminology . . . . . 164
11.3.1 Sets and Indices . . . . . 164
11.3.2 Grounded Planning Problem Instances . . . . . 166
11.3.3 Static Analysis . . . . . 167
11.4 Architecture of MIPS . . . . . 169
11.4.1 Heuristics . . . . . 169
11.4.2 Exploration Algorithms . . . . . 171
11.5 Temporal Planning . . . . . 173
11.5.1 Temporal Model . . . . . 173
11.5.2 Operator Dependency . . . . . 174
11.5.3 Critical Path Analysis . . . . . 176
11.5.4 Graphplan Distances . . . . . 177

11.5.5 Full Enumeration Algorithms . . . 178
11.5.6 Heuristic Search Enumeration . . . 180
11.5.7 Pruning Anomalies . . . 181
11.5.8 Arbitrary Plan Objectives . . . 181
11.6 Symmetry . . . 182
11.6.1 Static Symmetries . . . 183
11.6.2 Dynamic Symmetries . . . 184
11.6.3 Symmetry Reduction in MIPS . . . 185
11.7 Visualization . . . 186
11.8 Related Work . . . 188
11.8.1 Problem Classes and Methods . . . 188
11.8.2 Competing Planners . . . 189
11.8.3 Symbolic Model Checking based Planners . . . 191
11.9 Conclusions . . . 192

**IV Theorem Proving 193**

**12 Directed Automated Theorem Proving 195**
12.1 Introduction . . . 196
12.2 Functional Heuristic Search . . . 196
12.3 Heuristics for Automated Theorem Proving . . . 199
12.4 Isabelle . . . 200
12.5 Experiments . . . 201
12.6 Related Work . . . 205
12.7 Conclusions and Future Work . . . 208

**V Software Verification 209**

**13 Inferring Flow of Control 211**
13.1 Introduction . . . 212
13.1.1 Program Synthesis from Examples . . . 212
13.1.2 Programming in the Graphical User Interface . . . 212
13.2 Editing a First Example Trace . . . 213
13.3 The ID-Algorithm . . . 214
13.4 Customizing ID for Program Traces . . . 215
13.4.1 Naive Approach . . . 215
13.4.2 Pruning . . . 216
13.4.3 Selection of Example Data . . . 216
13.4.4 Query Complexity . . . 217
13.5 Transforming Automata into Structured Programs . . . 218
13.6 Conclusion and Discussion . . . 219

**14 Directed Explicit-State Model Checking 223**
14.1 Introduction . . . 224
14.2 Automata-based Model Checking . . . 225
14.2.1 Automata-theoretic Framework . . . 226

14.2.2 Search Algorithms . . . . . 227
14.2.3 The Model Checker SPIN . . . . . 227
14.2.4 Error Trails . . . . . 228
14.3 Heuristic Search Algorithms . . . . . 231
14.3.1 Depth-First, Breadth-First and Best-First Search . . . . . 231
14.3.2 Algorithm A* . . . . . 231
14.3.3 Iterative Deepening A* . . . . . 234
14.4 Search Heuristics for Safety Properties . . . . . 235
14.5 The HSF-SPIN Tool Set . . . . . 240
14.6 Safety Property Validation Experiments . . . . . 240
14.6.1 Shorter Trails and Computational Effort . . . . . 241
14.6.2 Heuristic Estimates . . . . . 243
14.6.3 Finding Errors where DFS fails . . . . . 244
14.6.4 IDA* and Bitstate Hashing . . . . . 245
14.7 Liveness Property Validation . . . . . 246
14.7.1 Classification of Never Claims . . . . . 247
14.7.2 Improved Nested Depth-First Search . . . . . 249
14.7.3 A* and Improved-Nested-DFS . . . . . 250
14.7.4 Correctness of INDFS . . . . . 251
14.8 Liveness Property Experiments . . . . . 254
14.8.1 INDFS for Validating Correctness . . . . . 254
14.8.2 INDFS for Error Detection . . . . . 255
14.9 Related Work . . . . . 256
14.10 Conclusion and Outlook . . . . . 258

**15 Trail-Directed Model Checking** **261**
15.1 Introduction . . . . . 262
15.2 Heuristic Search . . . . . 263
15.3 HSF-SPIN . . . . . 264
15.3.1 A First Example . . . . . 264
15.3.2 Compile- and Run-Time Options . . . . . 265
15.4 Improvement of Trails . . . . . 265
15.4.1 Hamming Distance Heuristic . . . . . 266
15.4.2 FSM Distance Heuristic . . . . . 267
15.4.3 Safety Errors . . . . . 267
15.4.4 Liveness Properties . . . . . 268
15.5 Experiments . . . . . 269
15.6 Conclusions . . . . . 270

**16 Partial-Order Reduction** **273**
16.1 Introduction . . . . . 274
16.2 Directed Model Checking . . . . . 275
16.2.1 General State Expanding Search Algorithm . . . . . 275
16.2.2 Algorithm A* . . . . . 275
16.2.3 Iterative-deepening A* . . . . . 276
16.2.4 Heuristic Estimates . . . . . 276
16.3 Partial Order Reduction . . . . . 277

16.3.1 Stuttering Equivalence of Labeled Transition Systems . . . . . . 277
16.3.2 Ample Set Construction for Safety $LTL_{-X}$ . . . . . . . . . . . . 278
16.3.3 Dynamically Checking C3 . . . . . . . . . . . . . . . . . . . . . . . . 279
16.3.4 Statically Checking C3 . . . . . . . . . . . . . . . . . . . . . . . . . 280
16.3.5 Hierarchy of C3 Conditions . . . . . . . . . . . . . . . . . . . . . . 281
16.3.6 Solution Quality and Partial Order . . . . . . . . . . . . . . . . . 281
16.4 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 282
16.4.1 Exhaustive Exploration . . . . . . . . . . . . . . . . . . . . . . . . 282
16.4.2 Error Finding with A* and Partial Order Reduction . . . . . . . . 283
16.4.3 Error Finding with IDA* and Partial Order Reduction . . . . . . . 284
16.4.4 Combined Effect of Heuristic Search and Partial Order . . . . . . 285
16.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 286

**VI Hardware Verification 287**

**17 Directed Symbolic Model Checking 289**
17.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 290
17.2 BDD Basics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 291
17.3 Model Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 291
17.3.1 Traditional Symbolic Model Checking . . . . . . . . . . . . . . . . 292
17.3.2 Other Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . 292
17.4 Directed Model Checking . . . . . . . . . . . . . . . . . . . . . . . . . . 293
17.4.1 BDDA* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
17.4.2 Heuristics and A* . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
17.4.3 Tailoring BDDA* for Model Checking . . . . . . . . . . . . . . . . 295
17.5 Inferring the Heuristic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 296
17.5.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 297
17.5.2 Refinement-Depth . . . . . . . . . . . . . . . . . . . . . . . . . . . 297
17.5.3 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 298
17.5.4 Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 298
17.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 301
17.7 Conclusion and Discussion . . . . . . . . . . . . . . . . . . . . . . . . . 302

**VII Route Planning 305**

**18 Localizing A* 307**
18.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 308
18.2 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 309
18.2.1 Traditional $A^*$ = Dijkstra + Re-weighting . . . . . . . . . . . . . 309
18.2.2 Invariance Condition . . . . . . . . . . . . . . . . . . . . . . . . . . 310
18.2.3 General-Node-Ordering $\boldsymbol{A}^*$ . . . . . . . . . . . . . . . . . . . . . . . 311
18.3 The Heap-Of-Heaps Data Structures . . . . . . . . . . . . . . . . . . . . 312
18.4 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 314
18.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 316
18.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 317

**19 Route Planning with GPS Traces** **319**
19.1 Introduction . . . 320
19.2 Travel Graph Construction . . . 320
19.2.1 Notation . . . 321
19.2.2 Algorithm of Bentley and Ottmann . . . 322
19.3 Statistical Map Inference . . . 323
19.3.1 Steps in the Map Refinement Process . . . 324
19.3.2 Map Segmentation . . . 325
19.3.3 Road Centerline Generation . . . 329
19.3.4 Lane Finding . . . 332
19.3.5 Experimental Results . . . 335
19.4 Searching the Map Graph . . . 336
19.4.1 Algorithm of Dijkstra . . . 336
19.4.2 Planar Graphs . . . 337
19.4.3 Frontier Search . . . 338
19.4.4 Heuristic Search . . . 338
19.5 Related Work . . . 339
19.6 Conclusions . . . 340

# Part I

# Foundations

# Paper 1

# Memory Limitation in Artificial Intelligence

Stefan Edelkamp
Institut für Informatik
Georges-Köhler-Allee, Geb. 51
79110 Freiburg, Germany
edelkamp@informatik.uni-freiburg.de

In *Peter Sanders, Uli Meyer, Job Sybeyn, ed., Memory Hierarchies*, Lecture Notes in Computer Science. Springer, 2002, to appear.

**Abstract**

This article surveys approaches to cope with limited memory for the design of algorithms and data structures in Artificial Intelligence, since many systems, e.g. in puzzle solving, two-player games, action and route planning, robotics, machine learning, data mining, logic programming and theorem proving, explore very large or even infinite implicitly given state spaces.

The main focus is on strategies for an improved time-space trade-off in case main memory becomes exhausted and include refined representation and exploration techniques, the introduction of additional search or control knowledge, and the explicit maintenance of secondary memory.

## 1.1 Introduction

Artificial Intelligence (AI) deals with structuring large amounts of data. As a very first example of an expert system [195], take the oldest known scientific treatise surviving from the ancient world, the surgical papyrus [45] of about 3000 BC. It discusses cases of injured men for whom a surgeon had no hope of saving and lay many years unnoticed until it was rediscovered and published for the New York Historical Society. The papyrus summarizes surgical observations of head wounds disclosing an inductive method for inference [119], with observations that were stated with title, examination, diagnosis, treatment, prognosis and glosses much in the sense that *if a patient has this symptom, then he has this injury with this prognosis if this treatment is applied.*

About half a century ago, pioneering computer scientists report the emergence of intelligence with machines that think, learn and create [326]. The prospects were driven by early successes in exploration. Samuel [312] wrote a checkers-playing program that was able to beat him, whereas Newell and Simon [277] successfully ran the general problem solver (GPS) that reduced the difference between the predicted and the desired outcome on different state-space problems. GPS represents problems as the task of transforming one symbolic expression into another, with a decomposition that fitted well with the structure of several other problem solving programs. Due to small available memories and slow CPUs, these and some other promising initial AI programs were limited in their problem solving abilities and failed to scale in later years.

There are two basic problems to overcome [257]: the *frame problem* – characterized as *the smoking pistol behind a lot of the attacks on AI* [78] – refers to all that is going on around the central actors, while the *qualification problem* refers to the host of qualifiers to stop an expected rule from being followed exactly. While [206] identifies several arguments of why intelligence in a computer is not a true ontological one, the most important reason for many drawbacks in AI are existing limits in computational resources, especially in memory, which is often too small to keep all information for a suitable inference accessible.

Bounded resources lead to a performance-oriented interpretation of the term intelligence: different to the Turing-test [341], programs have to show human-adequate or human-superior abilities in a competitive resource-constrained environment on a selected class of benchmarks. As a consequence even the same program can be judged to be more intelligent, when ran on better hardware or when given more time to execute. This competitive view settles; international competitions in data mining (e.g. KDD-Cup), game playing (e.g. Olympiads), robotics (e.g. Robo-Cup), theorem proving (e.g. CADE ATP), and action planning (e.g. IPC) call for highly performant systems on current machines with space and time limitations.

## 1.2 Hierarchical Memory

Restricted main memory calls for the use of *secondary memory*, where objects are either scheduled by the underlying operating system or explicitly maintained by the application program.

Hierarchical memory problems have been confronted to AI for many years. As an example, take the *garbage collector problem*. Minsky [264] proposes the first copying

garbage collector for LISP; an algorithm using serial secondary memory. The live data is copied out to a file on disk, and then read back in, in a contingous area of the heap space; [33] extends [264] to parallelize Prolog based on Warren's abstract machine, and modern copy collectors in *C++* [117] also refer to [264]. Moreover, garbage collection has a bad reputation for thrashing caches [198].

Access time graduates on current memory structures: processor register are better available than pre-fetched data, first-level and second level caches are more performant than main memory, which in turn is faster than external data on hard disks optical hardware devices and magnetic tapes. Last but not least, there is the access of data via local area networks and the Internet connections. The faster the access to the memorized data the better the inference.

Access to the next lower level in the memory hierarchy is organized in *pages* or *blocks*. Since the theoretical models of hierarchical memory differ e.g. by the amount of disks to be concurently accessible, algorithms are often ranked according to *sorting complexity* $\mathcal{O}(sort(N))$, i.e., the number of block accesses (I/Os) necessary to sort $N$ numbers, and according to *scanning complexity* $\mathcal{O}(scan(N))$, i.e., the number of I/Os to read $N$ numbers. The usual assumption is that $N$ is much larger than $B$, the block size. Scanning complexity equals $\mathcal{O}(N/B)$ in a single disk model. The first libraries for improved secondary memory maintainance are LEDA-SM [73] and TPIE[1]. On the other end, recent developments of hardware significantly deviate from traditional von-Neumann architecture, e.g., the next generation of Intel processors have three processor cache levels. Cache anomalies are well known; e.g. recursive programs like Quicksort often perform unexpectedly well when compared to the state-of-the art.

Since the field of improved cache performance in AI is too young and moving too quickly for a comprehensive survey, in this paper we stick to knowledge exploration, in which memory restriction leads to a *coverage problem*: if the algorithm fails to encounter a memorized result, it has to (re-)explore large parts of the problem space. Implicit exploration corresponds to explicit graph search in the underlying problem graph. Unfortunately, theoretical results in external graph search are yet too weak to be practical, e.g. $\mathcal{O}(|V|+\textit{sort}(|V|+|E|))$ I/Os for breadth-first search (BFS) [272], where $|E|$ is the number of edges and $|V|$ is the number of nodes. One additional problem in external single-source shortest path (SSSP) computations is the design of performant external priority queues, for which tournament-trees [227] serve as the current best.

Most external graph search algorithms include $\mathcal{O}(|V|)$ I/Os for restructuring and reading the graph, an inacceptable bound for implicit search. Fortunately, for sparse graphs efficient I/O algorithms for BFS and SSSP have been developed [339]. For example, on planar graphs, BFS and SSSP can be performed in $\mathcal{O}(\textit{sort}(|V|))$ time. For general BFS, the best known result is $\mathcal{O}\left(\sqrt{|V| \cdot \textit{scan}(|V|+|E|)} + \textit{sort}(|V|+|E|)\right)$ I/Os [261].

In contrast, most AI techniques improve internal performance and include refined state-space representations, increased coverage and storage, limited recomputation of results, heursitic search, control rules, and application-dependent page handling, but close connections in the design of internal space saving strategies and external graph search indicate a potential for cross-fertilization.

We concentrate on single-agent search, game playing, and action planning, since in these areas, the success story is most impressive. Single-agent engines optimally

[1] `http://www.cs.duke.edu/TPIE`

solve challenges like Sokoban [199] and Atomix [194], the 24-Puzzle [218], and Rubik's Cube [215]. Nowadays domain-independent action planners [100, 143, 181] find plans for very large and even infinite mixed propositional and numerical, metric and temporal planning problems. Last but not least, game playing programs challenge human supremacy for example in *Chess* [192], *American Checkers* [317], *Backgammon* [338], *Hex* [13], *Computer Amazons* [271], and *Bridge* [144].

## 1.3 Single-Agent Search

Traditional single-agent search challenges are puzzles. The "fruit-fly" is the NP-complete $(n^2 - 1)$–Puzzle popularized by Loyd and Gardner [137]. Milestones for optimal solutions were [319] for $n = 3$, [213] for $n = 4$, and [221] for $n = 5$. Other solitaire games that are considered to be challenging are the above mentionend PSPACE-hard computer games Sokoban and Atomix. Real-life applications include *number partitioning* [216], *graph partitioning* [123], *robot-arm motion planning* [171], *route planning* [115], and *multiple sequence alignment* [353].

Single-agent search problems are either given explicitly in form of a weighted directed graph $G = (V, E, w)$, $w : E \to \mathbb{R}^+$, together with one start node $s \in V$ and (possibly several) goal nodes $T \subseteq V$, or implicitly spanned by a quintuple $(\mathcal{I}, \mathcal{O}, w, \textit{expand}, \textit{goal})$ of initial state $\mathcal{I}$, operator set $\mathcal{O}$, weight function $w : \mathcal{O} \to \mathbb{R}^+$, successor generation function *expand*, and goal predicate *goal*. As an additional input, heuristic search algorithms assume an estimate $h : V \to \mathbb{R}^+$, with $h(t) = 0$ for $t \in T$. Since single-agent search can model Turing machine computations, it is undecidable in general [296].

*Heuristic search* algorithms traverse the re-weighted problem graph. Re-weighting sets the new weight of $(u, v)$ to $w(u, v) - h(u) + h(v)$. The total weight of a path from $s$ to $u$ according to the new weights differs from the old one by $h(s) - h(u)$. Function $h$ is *admissible* if it is a lower bound, which is equivalent to the condition that any path from the current node to the set of goal nodes in the re-weighted graph is of non-negative total weight. Since on every cycle the accumulated weights in the original and re-weighted graph are the same, the transformation cannot lead to negatively weighted cycles. Heuristic $h$ is called *consistent*, if $h(u) \leq h(v)+w(u, v)$, for all $(u, v) \in E$. Consistent heuristics imply positive edge weights.

The *A** algorithm [161] traverses the state space according to a cost function $f(n) = g(n) + h(n)$, where $h(n)$ is the estimated distance from state $n$ to a goal and $g(n)$ is the actual shortest path distance from the initial state. Weighted A* scales between the two extremes; *best-first search* with $f(n) = h(n)$ and BFS with $f(n) = g(n)$. State spaces are interpreted as implicitly spanned problem graphs, so that A* can be casted as a variant of Dijkstra's SSSP algorithm [80] in the re-weighted graph (cf. Fig. 1.1). In case of negative values for $w(u, v) - h(u) + h(v)$ shorter paths to already expanded nodes may be found later in the exploration process. These nodes are *re-opened*; i.e. re-inserted in the set of horizon nodes. Given an admissible heuristic, A* yields an optimal cost path. Despite the reduction of explored space, the main weakness of A* is its high memory consumption, which grows linear with the total number of generated states; the number of expanded nodes $|V'| << |V|$ is still large compared to the main memory capacity of $M$ states.

*Iterative deepening A** (IDA*) [213] is a variant of A* with a sequence of bounded depth-first-search (DFS) iterations. In each iteration IDA* expands all nodes having a

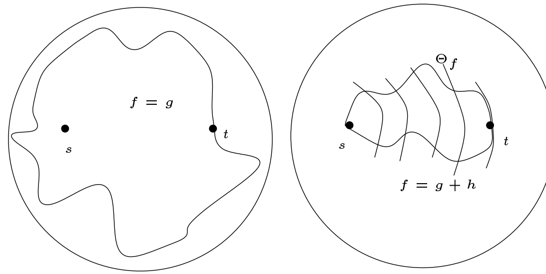


Figure 1.1: The effect of heuristics in A* and IDA* (right) compared to blind SSSP (left).

total cost not exceeding threshold $\Theta_f$, which is determined as the lowest cost of all generated but not expanded nodes in the previous iteration. The memory requirements in IDA* are linear in the depth of the search tree. On the other hand IDA* searches the tree expansion of the graph, which can be exponentially larger than the graph itself. Even on trees, IDA* may explore $\Omega(|V'|^2)$ nodes expanding one new node in each iteration. Accurate predictions on search tree growth [96] and IDA*'s exploration efforts [220] have been obtained at least for regular search spaces. In favor of IDA*, problem graphs are usually uniformly weighted with an exponentially growing search tree, so that many nodes are expanded in each iteration with the last one dominating the overall search effort.

As computer memories got larger, one approach was to develop better search algorithms *and* to use the available memory resources. The first suggestion was to memorize and update state information also for IDA* in form of a *transposition table* [305]. Increased coverage compared to ordinary hashing has been achieved by *state compression* and by *suffix lists*. State compression minimizes the state description length. For example the internal representation of a state in the 15-Puzzle can easily be reduced to 64 bits, 4 bits for each tile. Compression often reduces the binary encoding length to $\mathcal{O}(\log |V|)$, so that we might assume that for constant $c$ the states $u$ to be stored are assigned to a number $\phi(u)$ in $[1, \ldots, n = |V|^c]$. For the 15-Puzzle the size of the state space is $16!/2$, so that $c = 64/\lceil\log(16!/2)\rceil = 64/44 \approx 1.45$.

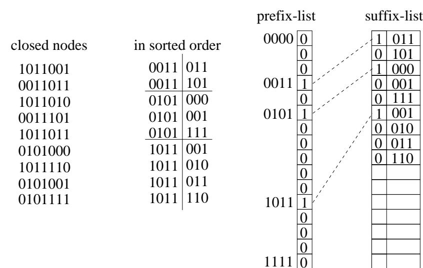


Figure 1.2: Example for suffix lists with $p = 4$, and $s = 3$.

Suffix lists [111] have been designed for external memory usage, but show a good space performance also for internal memorization. Let *bin*$(\phi(u))$ be the binary representation of an element $u$ with $\phi(u) \leq n$ to be stored. We split *bin*$(\phi(u))$ in $p$ high order bits and $s = \lceil\log n\rceil - p$ low order bits. Furthermore, $\phi(u)_{s+p-1}, \ldots, \phi(u)_s$ denotes the prefix of *bin*$(\phi(u))$ and $\phi(u)_{s-1}, \ldots, \phi(u)_0$ stands for the suffix of *bin*$(u)$. The suffix list consists of a linear array $P$ and of a two-dimensional array $L$. The basic idea of suffix lists is to store a common prefix of several entries as a single bit in $P$, whereas the distinc-

tive suffixes form a group within $L$. $P$ is stored as a bit array. $L$ can hold several groups with each group consisting of a multiple of $s + 1$ bits. The first bit of each $(s + 1)$-bit row in $L$ serves as a *group bit*. The first $s$ bit suffix entry of a group has group bit one, the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order, see Fig. 1.2. The space performance is by far better than ordinary hashing and very close to the *information theoretical bound*. To improve time performance to amortized $\mathcal{O}(\log |V|)$ for insertions and memberships, the algorithm buffers states and inserts checkpoints for faster prefix-sum computations.

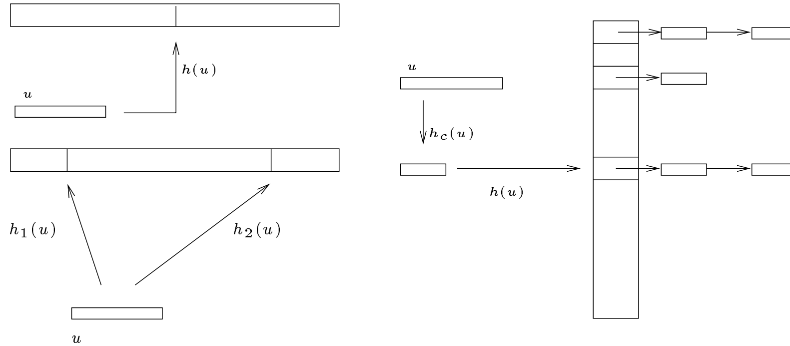


Figure 1.3: Single bit-state hashing, double bit-state hashing, and hash-compact.

*Bit-state hashing* [72] and *state compaction* reduce the state vector size to a selection of few bits allowing even larger table sizes. Fig. 2.4 illustrates the mapping of state $u$ via the hash functions $h$, $h_1$ and $h_2$ and compaction function $h_c$ to the according storage structures. This approach of *partial search* necessarily sacrifices completeness, but often yields shortest paths in practice [194]. While hash compact also applies to A*, single and double bit-state hashing are better suited to IDA* search [111], since the $f$ priority of a state and its predecessor pointer to track the solution, are mandatory for A*.

In regular search spaces, with a finite set of different operators to be applied, *Finite state machine (FSM) pruning* [337] provides an alternative for duplicate prediction in IDA*. FSM pruning pre-computes a string acceptor for move sequences that are guaranteed to have shorter equivalents; the set of *forbidden words*. For example, twisting two opposite sides of the Rubiks cube in one order, has always an equivalent in twisting them in the opposite order. This set of forbidden words is established by hash conflicts in a learning phase prior to the search and converted to a substring acceptor by the algorithm of Aho and Corasick [3]. Fig 1.4 shows an example to prune the search tree expansion in a regular Grid. The FSM enforces to follow the operators *up* (U), *down* (D), *left* (L), and *right* (R) along the corresponding arrows reducing the exponentially sized search tree expansion with $4^d$ states, $d > 0$, to the optimum of $d^2$ states. Suffix-tree automata [91] interleave FSM construction and usage.

In route planning based on *spatial partitioning* of the map, the heap-of-heap priority-queue data structure of Fig. 1.4 has effectivly been integrated in a localized A* algorithm [115]. The map is sorted according to the two dimensional physical layout and stored in form of small heaps, one per page, and one being active in main memory. To improve locality in the A* derivate, *deleteMin* is substituted by a specialized *deleteSome* operation that prefers node expansions with respect to the current page. The algorithm is shown both to be optimal and to significantly reduce page faults counter-balanced with a slight increase in the number of node expansions. Although locality information is ex-

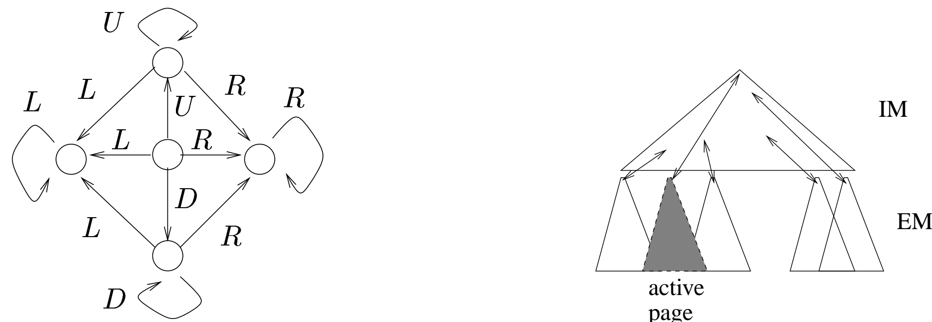


Figure 1.4: The finite state machine to prune the Grid (left) and the heap-of-heap data structure for localized A* (right). The main and the active heap are in internal memory (IM), while the others reside on external memory (EM).

ploited, parts of the heap-of-heap structure may be substituted by provably I/O efficient data structures like *buffer trees* [15] or *tournament trees* [227].

Most *memory-limited search* algorithms base on A*, and differ in the caching strategies when memory becomes exhausted. *MREC* [324] switches from *A** to *IDA** if the memory limit is reached. In contrast, *SMA** [311] reassigns the space by dynamically deleting a previously expanded node, propagating up computed $f$-values to the parents in order to save re-computation as far as possible. However, the effect of node caching is still limited. An adversary may request the nodes just deleted. The best theoretical results on search trees are $O(|V'| + M + |V'|^2/M)$ node expansions in the *MEIDA** search algorithm [88]. The algorithm works in two phases: The first phase fills the doubly-ended priority queue $D$ with at most $M$ nodes in IDA* manner. These nodes are expanded and re-inserted into the queue if they are *safe*, i.e., if $D$ is not full and the $f$-value of the successor node is still smaller than the maximal $f$-value in $D$. This is done until $D$ eventually becomes empty. The last expanded node then gives the bound for the next IDA* iteration. Let $E(i)$ be the number of expanded nodes in iteration $i$ and $R(i) = E(i) - E(i-1)$ the number of newly generated nodes in iteration $i$. If $l$ is the last iteration then the number of expanded nodes in the algorithm is $\sum_{i=1}^{l} i \cdot R(l-i+1)$. Maximizing $\sum_{i=1}^{l} i \cdot R(l-i+1)$ with respect to $R(1) + \ldots + R(l) = E(l) = |V'|$, and $R(i) \geq M$ for fixed $|V'|$ and $l$ yields $R(l) = 0$, $R(1) = |V'| - (l-2)M$ and $R(i) = M$, for $1 < i < l$. Hence, the objective function is maximized at $-Ml^2/2 + (|V'| + 3M/2)l - M$. Maximizing for $l$ yields $l = |V'|/M + 3/2$ and $O(|V'| + M + |V'|^2/M)$ nodes in total.

*Frontier search* [217, 222] contributes to the observation that the newly generated nodes in any graph search algorithm form a connected horizon to the set of expanded nodes, which is omitted to save memory. The technique refers to Hirschberg's linear space divide-and-conquer algorithm for computing maximal common sequences [176]. In other words, frontier search reduces a $(d+1)$-dimensional search problem into a $d$-dimentional one. It divides into three phases: in the first phase, a goal $t$ with optimal cost $f^*$ is searched; in the second phase the search is re-invoked with bound $f^*/2$; and by maintaining shortest paths to the resulting fringe the intermediate state $i$ from $s$ to $t$ is detected, in the last phase the algorithm is recursively called for the two subproblems from $s$ to $i$, and from $i$ to $t$. Fig. 1.5 depicts the recursion step and indicates the necessity to store virtual nodes in directed graphs to avoid falling back behind the search frontier, where a node $v$ is called *virtual*, if $(v, u) \in E$, and $u$ is already expanded.

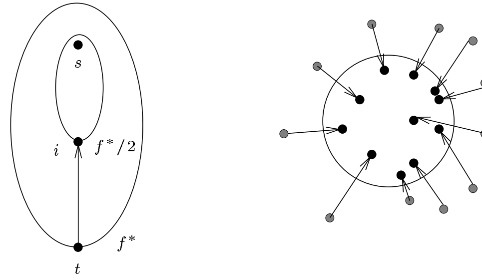


Figure 1.5: Divide step in undirected frontier search (left) and backward arc look-ahead in directed frontier search (right).

Many external exploration algorithms perform variants of frontier search. In the $\mathcal{O}(|V'| + sort(|V'| + |E'|))$ I/O algorithm of Munagala and Ranade [272] the set of visited lists is reduced to one additional layer. In difference to the internal setting above, this algorithm performs a complete exploration and uses external sorting for duplicate elimination.

Large *dead-end recognition tables* [199] are best built in sub-searches of problem abstractions and avoid non-progressing exploration. Fig. 1.6 gives an example of bootstrapping dead-end patterns by expansion and decomposition in the *Sokoban* problem, a block-sliding game, in which blocks are only to be pushed from an accessible side. Black ball patterns are found by simple recognizers, while gray ball patterns are inferred in bottom-up fashion. Established sub-patterns subsequently prune exploration.

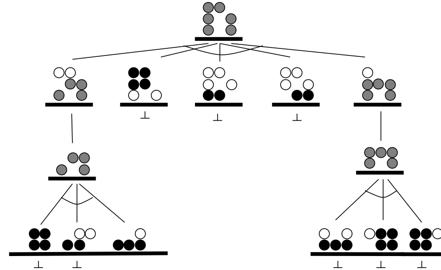

Figure 1.6: Bootstrapping to build dead-end recognition tables.

Another approach to speed up the search is to look for more accurate estimates. With a better heuristic function the search will be guided faster towards the goal state and has to deal with less of nodes to be stored. Problem-dependent estimates may have a large overhead to be computed for each encountered state, calling for a more intelligent usage of memory. *Perimeter Search* [82] is an algorithm that saves a large table in memory which contains all nodes that surround the goal node up to a fixed depth. The estimate is then defined as the distance of the current state and the closest state in the perimeter.

## 1.4 Action Planning

Domain-independent action planning [6] is one of the central problems in AI, which arises, for instance, when determining the course of action for a robot. Problem domains and instances are usually specified in a general domain description language

(PDDL) [258, 131]. Its "fruit-flies" are *Blocks World* and *Logistics*. Planning has effectively been applied for instance in *robot navigation* [154], *elevator scheduling* [211], and *autonomous spacecraft control* [135].

A classical (grounded) *Strips planning problem* [126] is formalized as a quadruple $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$, with $\mathcal{S} \subseteq 2^F$ being the set of states, $2^F$ being the power set of the set of propositional atoms $F$, $\mathcal{I} \in \mathcal{S}$, $\mathcal{G} \subseteq \mathcal{S}$, and $\mathcal{O}$ being the set of operators that transform states into states. A state $S \in \mathcal{S}$ is interpreted by the conjunct of its atoms. Operator $o = (P, A, D) \in \mathcal{O}$ is specified with its precondition, add and delete lists, $P, A, D \subseteq F$. If $P \subseteq S$, the result $S' \in \mathcal{S}$ of an operator $o = (P, A, D)$ applied to state $S \in \mathcal{S}$ is defined as $S' = (S \setminus D) \cup A$. *Mixed propositional and numerical planning* [131] take $\mathcal{S} \subseteq 2^F \times \mathbb{R}^k$, $k > 0$, as the set of states, *temporal planning* includes a special variable *total-time* to fix action execution time, and *metric planning* optimizes an additionally specified objective function. Strips planning is PSPACE-complete [53] and *mixed propositional and numerical planning* is undecidable [170].

Including a non-deterministic choice on actions effects is often used to model uncertainty of the environment. *Strong plans* [63], are plans that guarantee goal achievement despite all non-determinism. *Strong plans* are complete compactly stored state-action tables, that can be best viewed as a controller, that applies certain actions depending on the current state. In contrast, in *conformant planning* [62] a plan is a simple sequence of actions, that is successful for all non-deterministic behaviours. Planning with partial observability interleaves action execution and sensing. In contrast to the successor set generation based on action application, observations correspond to "And" nodes in the search tree [32]. Both conformant and partial observable planning can be casted as a deterministic traversal in *belief space*, defined as the power set $2^{\mathcal{S}}$ of the original one planning state space $\mathcal{S}$ [40]. Belief spaces and complete state-action tables are seemingly too large to be explicitly stored in main memory, calling for refined internal representation or fast external storage devices.

In *probabilistic planning* [298], different action outcomes are assigned to a probability distribution and resulting plans/policies correspond to complete *state-action tables*. Probabilistic planning problems are often modeled as *Markov decision process* (MDPs) and mostly solved either by policy or value iteration, where the latter invokes to successive updates to Bellmann's equation. The complexity of probabilistic planning with partial observability is $PP^{NP}$-complete [273]. Different caching strategies for solving larger partial observable probabilistic planning problems are studied in [251], with up to substantial CPU time savings for application dependent caching schemes.

Early planning approaches in Strips planning were able to solve only small Strips problems, e.g., to stack five blocks, but planning graphs, SAT-encodings, as well as heuristic search have changed the picture completely. *Graphplan* [39] constructs a layered planning graph containing two types of nodes, action nodes and proposition nodes. In each layer the preconditions of all operators are matched, such that *Graphplan* considers instantiated actions at specific points in time. *Graphplan* generates partially ordered plans to exhibit concurrent actions and alternates between two phases: *graph extension* to increase the search depth and *solution extraction* to terminate the planning process. *Satplan* [204] simulates a BFS according to the binary encoding of planning states, with a standard representation of Boolean formulae as a conjunct of clauses.

In current heuristic search planning, relaxed plans [181] and pattern databases (PDB) [95] turn out to be best (cf. Fig. 1.7). The *relaxed planning heuristic* generates ap-

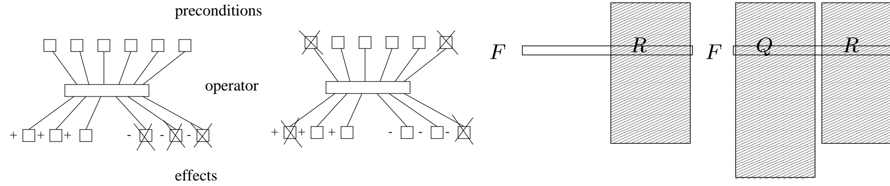


Figure 1.7: Operator abstractions for the relaxed planning and the pattern database heuristic (left); single and disjoint PDB for subsets $R$ and $Q$ of all atoms $F$ (right).

proximate plans to the simplified planning problem, in which negative effects have been omitted in all operators, and is computed in a combined forward and backward traversal for each encountered state. The first phase determines a fix point on the set of reachable atoms similar to *Graphplan*, while the second phase greedily extracts the approximate plan length as a heursitic estimate.

*Pattern databases* (PDB) [75] are constant-time look-up tables generated by completely explored problem abstractions. Subgoals are clustered and for all possible combination of these subgoals in every state of the problem, the relaxed problem is solved without looking on the other subgoals. Fig. 1.7 illustrates PDB construction. The planning state is represented as a set of propositional atoms $F$, and the operators are projected to one or several disjoint subsets of $F$. The simplified planning problem is completely explored with any SSSP algorithm, starting from the set of goal states. The abstract states are stored in large hash tables together with their respective goal distance, to serve as heuristic estimates for the overall search. Retrieved values of different databases are either maximized or added. PDBs optimally solved the *15-Puzzle* [75] and *Rubik's Cube* [215]. A space-time trade-off for PDB is analyzed in [174]; PDB size is shown to be inversely correlated to search time. Since search time is proportional to the number of expanded nodes $N'$ and PDB-size is proportional to $M$, PDBs make very effective use of main memory. Finding good PDB abstractions is not immediate. The general search strategy for optimal-sized PDBs [173] applies to a large set of state-space problems and uses IDA* search tree prediction formula [220] as a guidiance. A general bin-packing scheme to generate disjoint PDBs [218] is proposed [95]. Recall that disjoint PDBs generate very fast optimal solutions to the 24-Puzzle.

Some planners cast planning as *model checking* [145] and apply *binary decision diagrams* (BDDs) [50] to reduce space consumption. BDDs are compact acyclic graph data structures to represent and efficiently manipulate Boolean functions; the nodes are labeled with Boolean variables with two outgoing edges corresponding to the two possible outcomes when evaluating a given assignment, while the 0- or 1-sink finally yield the determined result. *Symbolic exploration* refers to *Satplan* and realizes a BFS exploration on BDD representations of sets of states, where a state is identified by its characteristic function. Given the represented set of states $S_i(x)$ in iteration $i$ and the represented transtion relation $T$ the successor set $S_{i+1}(x)$ is computed as $S_{i+1}(x) = \exists x \ (S_i(x) \wedge \ T(x,x'))[x/x']$. In contrast to *conjunctive partitioning* in hardware verification [263], for a refined image computation in symbolic planning *disjunctive partitioning* of the transition function is required: if $T(x,x') = \vee_{o \in \mathcal{O}} o(x,x')$ then $S_{i+1}(x) = \exists x \ (S_i(x) \wedge \vee_{o \in \mathcal{O}} o(x,x'))[x/x'] = \vee_{o \in \mathcal{O}} (\exists x \ (S_i(x) \wedge \ o(x,x')))[x/x']$.

Symbolic heuristic search maintains the search horizon *Open* and the heursitic estimate $H$ in compact (monolithical or partitioned) form. The algorithm BDDA* [112]

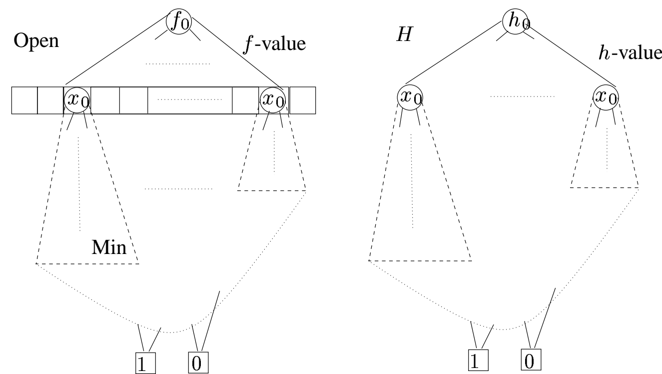


Figure 1.8: Symbolic heuristic search A* search with symbolic priority queue and estimate.

steadily extracts, evaluates, expands and re-inserts the set of states *Min* with minimum $f$-value (cf. Fig. 1.8). For consistent heursitics, the number of iterations in BDDA* can be bounded by the square of the optimal path length.

Although *algebraic decision diagrams* (ADDs), that extend BDDs with floating point labeled sinks, achieve no improvement to BDDs to guide a symbolic exploration in the 15-Puzzle [158], generalization for probabilistic planning results in a remarkable improvement to the state-of-the-art [124]. Pattern databases and symbolic representations with BDDs can be combined to create larger look-up tables and improved estimates for both explicit and symbolic heuristic search [99]. In *conformant planning* BDDs also apply best, while in this case heuristics have to trade information for exploration gain [31].

Since BDDs are also large graphs, improving memory locality has been studied e.g. in the breadth-first synthesis of BDDs, that constructs a diagram levelwise [193]. There is a trade-off between memory overhead and memory access locality, so that hybrid approaches based on context switches have been explored [351]. Efficiency analysis show that BDD reduction of a decision diagram $G$ can be achieved in $\mathcal{O}(sort(|G|))$ I/Os, while Boolean operator application to combine two BDDs $G_1$ and $G_2$ can be performed in $\mathcal{O}(sort(|G_1| + |G_2|))$ I/Os [14].

Domain specific information can control forward chaining search [17]. The proposed algorithm progresses first order knowledge through operator application to generate an extended state description and may be interpreted as a form of parameterized FSM pruning.

Another space efficient representation for depth-first exploration of the planning space is a *persistent search tree* [16], storing and maintaining the set of instantiations of planning predicates and functions. Recall that persistent data stuctures only store differences of states, and are often used for text editors or version management systems providing fast and memory-friendly random access to any previously encountered state.

Mixed propositional, temporal and numerical planning aspects call for plan schedules, in which each action is attached to a fixed time-interval. In contrast to ordinary scheduling the duration of an action can be state-dependent. The currently leading approaches[2] are an interleaved plan generator and optimal (*PERT*) scheduler of the imposed causal structure (MIPS), and a local search engine on planning graphs, optimizing plan quality by deleting and adding actions of generated plans governed by Lagrange multipliers (LPG).

[2] www.dur.ac.uk/d.p.long/competition.html

Static analysis of planning domains [129] leads to a general efficient state compression algorithm [101] and is helpful in different planners, especially in BDD engines. *Generic type* analysis of domain classes [244] drives the design of hybrid planners, while different forms of symmetry reduction based on object isomorphisms, effectively shrink exploration space [132]: *generic types* exploit similarities of different domain structures, and symmetry detection utilizes the parametric description of domain predicates and actions.

Action planning is closely related to *error detection* in software [105] and hardware designs [303], where systems are modeled as state transition graphs of synchronous or asynchronous systems and analyzed by reasoning about properties of states or paths. As in planning, the central problem is overcoming combinatorial explosion; the number of system states is often exponential in the number of state variables. The transfer of technology is rising: *Bounded model checking* [36] exports the *Satplan* idea to error detection, *symbolic model checking* has lead to BDD based planning, while *directed model checking* [105] matches with the success achieved with heuristic search planning.

One approach that has not yet been carried over is *partial order reduction* [241], which compresses the state space by of avoiding concurrent actions, thus reducing the effective branching factor. In difference to FSM pruning, partial ordering sacrifices optimality, detects neccessary pruning conditions on the fly, and utilizes the fact that the state space is composed by the cross product of smaller state spaces containing many local operators.

## 1.5 Game Playing

One research area of AI that has ever since dealt with given resource limitations is game playing [316]. Take for example a *two-payer zero-sum game* (with perfect information) given by a set of states $\mathcal{S}$, move-rules to modify states and two players, called Player 0 and Player 1. Since one player is active at a time, the entire state space of the game is $\mathcal{Q} = \mathcal{S} \times \{0, 1\}$. A game has an initial state and some predicate *goal* to determine whether the game has come to an end. We assume that every path from the initial state to a final one is finite. For the set of goal states $\mathcal{G} = \{s \in \mathcal{Q} \mid goal(s)\}$ we define an evaluation function $v : \mathcal{G} \to \{-1, 0, 1\}$, $-1$ for a lost position, $1$ for a winning position, and 0 for a draw. This function is extended to $\hat{v} : \mathcal{Q} \to \{-1, 0, 1\}$ asserting a game theoretical value to each state in the game. More general settings are *multi-player games* and *negotiable games* with incomplete information [300].

DFS dominates game playing and especially computer chess [254], for which [167] provides a concise primer, including *mini-max search*, *$\alpha\beta$ pruning*, *minimal-window* and *quiescence search* as well as *iterative deepening*, *move ordering*, and *forward pruning*. Since game trees are often too large to be completely generated in time, static evaluation functions assert numbers to root nodes of unexpored subtrees. Fig. 1.9 illustrates a simple *mini-max game tree* with leaf evaluation, and its reduction by $\alpha\beta$ search and move ordering. In a game tree of height $h$ with *branching factor* $b$ the minimal traversed part tree reduces from size $\mathcal{O}(b^h)$ to $\mathcal{O}(\sqrt{b^h})$. Quiescence search extends evaluation beyond exploration depth until a quiescent position is reached, while forward pruning refers to different unsound cut-off techniques to break full-width search. Minimal-window search is another inexact approximation of $\alpha\beta$ with higher cut-off rates.

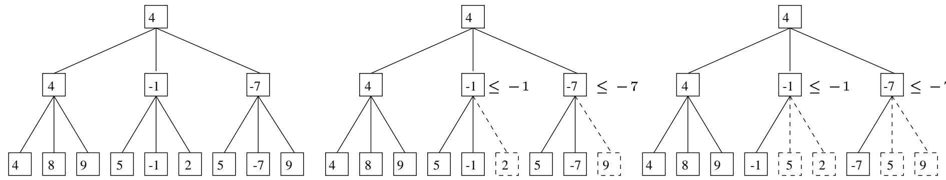


Figure 1.9: Mini-max game search tree pruned by $\alpha\beta$ and additional move ordering.

As in single-agent search, transposition tables are memory-intense containers of search information for valuable reuse. The stored move always provides information, but the memorized score is applicable only if the nominal depth does not exceed the value of the cached draft. Since the early 1950s, from the "fruit-fly"-status, chess has advanced to one of the main successes in AI, resulting in the defeat of the human-world champion in a tournament match. DeepThought [253] utilized IBM's DeepBlue architecture for a massive-parallelized, hardware-oriented $\alpha\beta$ search scheme, evaluating and storing billions of nodes within a second, with a fine-tuned evaluation function and a large, man-made and computer-validated opening book.

*Nine-Men-Morris* has been solved with huge *endgame databases* (EDB) [139], in which every state after the initial placement has been asserted to its *game-theoretical value*. The outcome of a complete search is that the game is a draw. Note that the DeepBlue chess engine is also known to have held complete EDB on the on-chip memories.

*Four Connect* has been proven to be a win for the first player in optimal play using a knowledge-based approach [7] and mini-max-based *proof number search* (PNS) [8], that introduces the third value *unknown* into the game search tree evaluation. PNS has a working memory requirement linear in the size of the search tree, while $\alpha\beta$ requires only memory linear to the depth of the tree. To reduce memory consumption [8], solved subtrees are removed, or leveled execution is performed. PNS also solved *GoMoku* [9], where the search tree is partitioned into a few hundred subtrees, externally stored and combined into a final one [9]. Proof-Set Search is a recent improvement to PNS, that saves node explorations for a more involved memory handling [270].

*Hex* is another PSPACE complete board game invented by the Danish mathematician Hein [48]. Since the game can never result in a draw it is easy to prove that the game is won for the first player to move, since otherwise he can adopt the winning strategy of the second player to win the game. The current state-of-the-art program *Hexy* uses a quite unusual approach electrical circuit theory to combine the influence of sub-positions (virtual connections) to larger ones [13].

*Go* has been addressed by different strategies. One important approach [268] with exponential savings in some endgames uses a divide-and-conquer method based on *combinatorial game theory* [29] in which some board situations are split into a sum of local games of tractable size. *Partial order bounding* [269] propagates relative evaluations in the tree and has also been shown to be effective in Go endgames. It applies to all mini-max searchers, such as $\alpha\beta$ and PNS.

An alternative to $\alpha\beta$ search with minimax evaluation is *conspiracy number search* (CNS) [255]. The basic idea of CNS is to search the game tree in a manner that at least $c > 1$ leaf values have to change in order to change the root one. CNS has been successfully applied to chess [247, 314].

Memory limitation is most apparent in the construction of EDBs [343]. Different to analytical machine learning approaches [320], that construct an explanation why the concept being learned works for positive learning examples – to be stored in operational form for later re-use in similar cases – EDBs do not discover general rules for infallible play, but are primary sources of information for the game-theoretical value of the respective endgame positions. The major compression schemes for positions without pawns use symmetries along the axes of the chess board.

EDB can also be constructed with symbolic, BDD-based exploration [20, 98], but an improving integration of symbolic EDBs in game playing has still to be given. Some combinatorial chess problems like the total number of 33,439,123,484,294 complete Knight's tours have been solved with the compressed representation of BDDs [242].

For EDBs to fit into main memory the general principle is to find efficient encodings. For external usage run-length encoding suits best for output in a final external file [231]. Huffman encodings [70] are further promising candidates. Thereby, modern game playing programs quickly become I/O bound, if they probe external EDBs not only at the root node. In checkers [316] the distributed generation of a very large EDBs has given the edge in favor to the computer. Schaeffer's checker program *Chinook* [315] has perfect EDB information for all checker positions involving eight or fewer pieces on the board, a total of 443,748,401,247 positions generated in large *retrograde analysis* using a network of workstations and various high-end computers [231]. Commonly accessed portions of the database are pre-loaded into memory and have a greater than 99% hit rate with a 500MB cache [225]. Even with this large cache, the sequential version of the program is I/O bound. A parallel searching version of *Chinook* further increased the I/O rate such that computing the database was even more I/O intensive than running a match.

Interior-node recognition [328] is another memorization technique in game playing that includes game-theoretical information in form of score values to cut-off whole subtrees for interior node evaluation in $\alpha\beta$ search engines. Recognizers are only invoked, if transposition table lookups fail. To enrich the game theoretical information, *material signatures* are helpful. The memory access is layered. Firstly, appropriate recognizers are efficiently detected and selected before a lookup into an EDB is performed [166].

## 1.6 Other AI Areas

An apparent candidate for hierarchical memory exploitation is *data or information mining* [310]; the process of inferring knowledge from very large databases. *Web mining* [223] is data mining in the Internet where *intelligent internet systems* [236] consider user modeling, information source discovering and information integration. Classification and clustering in data mining [266] links to the wide range of *machine learning* techniques [232] with *decision-tree* and *statistical methods*, *neural networks*, *genetic algorithms*, *nearest neighbor search* and *rule induction*. Association rules are implications of the form $X \Rightarrow I$ with $I$ being a binary attribute. Set $X$ has support $s$, if $s\%$ of all data is in $X$, whereas a rule $X \Rightarrow I$ has confidence $c$, if $c\%$ of all data that are in $X$ also obey $I$. Given a set of transactions $D$, the problem is to generate all association rules that have user-specified minimum support and confidence. The main association rule induction algorithm is *AIS* [1]. For fast discovery, the algorithm was improved in *Apriori* [2]. The first pass of the algorithm simply counts the number of occurences of each item to determine

itemsets of cardinality 1 with minimum support. In the $k$-th pass the itemset with $(k-1)$ elements and minimum support of phase $(k-1)$ are used to generate a candidate set, which by scanning the database yields the support of the candidate set and the $k$-itemset with minimum support. The running time of *Apriori* is $\mathcal{O}(|C| \cdot |D|)$, where $|D|$ is the size of the database and $|C|$ the total number of generated candidates. Even advanced association rule inference requires substantial processing power and main memory [348]. An example to hierarchical memory usage is a distributed rule discovery algorithm [57].

*Case-based reasoning*(CBR) [203] systems integrate database storage technology into knowledge representation systems. CBR systems store previous experiences (cases) in memory and in order to solve new problems, i) retrieve similar experience about similar situation from memory ii) complete or partial re-use or adapt the experience in the context of the new situation, iii) store new experience in memory. We give a few examples that are reported to explicitly use secondary memory. Parka-DB [334] is a knowledge base with a reduction in primary storage with 10% overhead in time, decreasing the load time by more than two orders of magnitude. Framer [155] is a disk-based object-oriented knowledge based system, whereas Thenetsys [285] is a semantic network system that employs secondary memory structure to transfer network nodes from the disk into main memory and vice versa.

*Automated theorem proving* procedures draw inferences on a set of clauses $\Gamma \to \Delta$, with $\Gamma$ and $\Delta$ as multisets of atoms. A top-down proof creates a proof tree, where the node label of each interior node corresponds to the conclusion, and the node labels of its children correspond to the premises of an inference step. Leaves of the proof tree are axioms or instances of proven theorems. A *proof state* represents the outer fragment of a proof tree: the top-node, representing the goal and all leaves, representing the subgoals of the proof state. All proven leaves can be discharged, because they are not needed for further proof search. If all subgoals have been solved, the proof is sucessful. Similar to action planning, proof-state based automated theorem proving spans large and infinite state spaces. The overall problem is undecidable and can be tackled by user invention and implicit enumeration only. While polynomial decision procedures exists for restricted classes [256], first general heuristic search algorithms to accelerate exploration have been proposed [106].

## 1.7 Conclusions

The spectrum of research in memory limitations algorithms for representing and exploring large or even infinite problem spaces is enormous and encompasses large subareas of AI. We have seen alternative approaches to exploit and memorize problem specific knowledge and some schemes that explicitly schedule external memory. Computational trade-offs under bounded resources become increasingly important, as e.g. a recent issue of *Artificial Intelligence* [191] with articles on recursive conditioning, algorithm portfolios, anytime algorithms, continual computation, and iterative state space reduction indicates. Improved design of hierarchical memory algorithms, probably special-tailored to AI exploration, are apparently needed.

Nevertheless, there is much more research, sensibility, and transfer of results needed, as two feedbacks of German AI researchers illustrate. For the case of external algorithms, Bernhard Nebel [276] mentions that current memory sizes of 256MB up to several GB

make the question of refined secondary memory access no longer that important. This argument neglects that even by larger amount of main memory the latency gap still rises, and that with current CPU speed, exploration engines often exhaust main memory in less than a few minutes.

For the case of processor performance tuning, action execution in robotics has a high frequency of 10-20 Hz, but there is almost no research in improved cache performance: Wolfram Burgard [52] reports some successes by restructuring loops in one application, but has also seen failures for hand-coded assembler inlines to beat the optimized compiler outcome in another.

The ultimative motivation for an increased research in space limitations and hierarchical memory usage in AI is its inspiror, the human brain, with an hierarchical layered organization structure, including ultra short time working memory, as well as short and long time memorization capabilities.

# Paper 2

# Theory and Practice of Time-Space Trade-Offs in Memory Limited Search

Stefan Edelkamp
Institut für Informatik
Georges-Köhler-Allee, Geb. 51
79110 Freiburg, Germany
edelkamp@informatik.uni-freiburg.de

Ulrich Meyer
Max-Plank-Institut für Informatik
Stuhlsatzenhausenweg 85
66123 Saarbrücken, Germany
umeyer@mpi-sb.mpg.de

In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science. Springer, pages 169–184, 2001.

**Abstract**

Having to cope with memory limitations is an ubiquitous issue in heuristic search. We present theoretical and practical results on new variants for exploring state-space with respect to memory limitations.

We establish $O(\log n)$ minimum-space algorithms that omit both the open and the closed list to determine the shortest path between every two nodes and study the gap in between full memorization in a hash table and the information-theoretic lower bound. The proposed structure of suffix-lists elaborates on a concise binary representation of states by applying bit-state hashing techniques. Significantly more states can be stored while searching and inserting $n$ items into suffix lists is still available in $O(n \log n)$ time. Bit-state hashing leads to the new paradigm of partial iterative-deepening heuristic search, in which full exploration is sacrificed for a better detection of duplicates in large search depth. We give first promising results in the application area of communication protocols.

**IDA***$(s)$
    Push$(S, s, h(s))$; $U \leftarrow h(s)$
    **while** $(U \neq \infty)$
        $U \leftarrow U'$; $U' \leftarrow \infty$
        **while** $(S \neq \emptyset)$
            $(u, f(u)) \leftarrow$ Pop$(S)$
            **if** $(goal(u))$ **return** $(u, f(u))$
            **for all** $v$ **in** $\Gamma(u)$
                **if** $(f(u) + w(u,v) - h(u) + h(v) > U)$
                    **if** $(f(u) + w(u,v) - h(u) + h(v) < U')$
                        $U' \leftarrow f(u) + w(u,v) - h(u) + h(v)$
                **else**
                    Push$(S, v, f(u) + w(u,v) - h(u) + h(v))$

Table 2.1: The IDA* Algorihm implemented with a Stack.

## 2.1 Introduction

Heuristic search in large problem spaces inherently calls for algorithms capable of running under restricted memory. We present new data structures and algorithms that face the memory vs. duplication elimination problem that still arises even if the exploration is directed. The class of *memory-restricted search algorithms* has been developed under this aim. The framework imposes an absolute upper bound on the total memory the algorithm may use, regardless of the size of the problem space. If the number of nodes with distance value smaller than the optimal solution path length is larger than this memory bound, storing the entire list of visited nodes is no longer possible.

*Iterative deepening A**, IDA* for short [213], has proven effective to successively search the problem graph with bounded DFS traversals according to an increasing threshold for the tentative values. IDA* consumes space linear in the solution length. It does not use additionally available memory and traverses all generating paths. Unfortunately, the number of paths in a graph might be exponentially larger than the number of nodes such that the design of informative consistent heuristics and duplicate elimination remains essential. If all merits are distinct, IDA* expands a quadratic number of nodes in the worst case. Although iterative deepening is limited to small integral weights it performs well in practice. Table 2.1 depicts a possible implementation of IDA* in pseudo-code: $S$ is a stack for backtracking, $U$ is the current threshold, and $U'$ the threshold for the next iteration. The value $w(u, v)$ is the weight of the transitition $(u, v)$, $h(u)$ and $f(u)$ is the heuristic estimate and combined merit for node $u$, respectively.

Pattern data-bases [75] are a general tool to improve the estimate that can cope with complex subproblem interactions. A solution preserving relaxation of the search problem is traversed prior to the search and the goal distances of all abstract states are kept as lower bound estimats for the overall problem within a large hash table. However, the application of this pre-compilation technique is limited to suitable domain abstractions that yield better results than on-line computations as findings in protocol verification [109], AI-planning [95], and selected single-agent problems [194] indicate. Therefore, to lessen

memory consumption according to a large number of states is still a problem.

Transposition tables are used to store and improve the distances until the memory bound has been reached [305]. However, when the memory is exhausted, IDA*'s time consumption is often stinged by uncaught duplicates.

Different node caching strategies have been applied: MREC [324] switches from A* to IDA* if the memory limit is reached. In contrast, SMA* [311] reassigns the space by dynamically deleting a previously expanded node, propagating up computed $f$-values to the parents in order to save re-computation as far as possible. However, the effect of node cashing is still limited. An adversary may request the nodes just deleted.

The paper is aimed to close this gap and is structured as follows: The first section gives an $O(\log n)$-space algorithm to search for the shortest path in graphs with uniform or small weights, with $n$ being the total number of nodes in the problem graph. Suffix lists are a data structure for maximizing the number of stored states according to a given memory limit. The achieved result is compared to ordinary hashing and a derived information-theoretic bound. Bit-state state compaction, sequential hashing and partial search can substitute the transposition table of IDA* with a bit-vector table. Thereby, it is possible to detect more duplicates in the space while increasing the depth of the search. We give promising experimental results in validating an industrial communication protocol.

## 2.2 Minimum Space Algorithms

First of all, we might ask for the limit of space reduction. Given a graph with $n$ nodes we are interested in algorithms that compute the BFS-level and shortest paths of all nodes and either consume as little working space as possible or perform faster if more space is available. In addition, we assume that the algorithms are not allowed to modify the input during the exection.

The similar problems of node reachability (i.e., determine whether there any path between two nodes) and graph connectivity have been efficiently solved for the same restricted memory setting using random walk strategies [121, 122]. However, we are not aware of any equivalent results for BFS and shortest paths. In the following we will devise an $O(\log n)$ space algorithm for BFS and shortest paths with small integer weights. The principle is similar to the simulation of nondeterministic Turing machines [313].

### 2.2.1 Divide-And-Conquer BFS

To compute the breadth-first-level for each node, with very limited space, we may use a DAC strategy *Path* that reports if there is a path from $a$ to $b$ with $l$ edges. If $l$ equals 1 and there is an edge from $a$ to $b$ then the procedure returns true. Otherwise, for each node index $j$, $1 \leq j \leq n$, we recursively determine *Path*$(a, j, \lceil l/2 \rceil)$ and *Path*$(j, b, \lfloor l/2 \rfloor)$. If both exist the returned value is true, compare Table 2.2. The recursion stack has to store at most $O(\log n)$ frames each of which contains $O(1)$ integers. Hence the space complexity is $O(\log n)$. However, this has to be paid with a time complexity of $O(n^{3+\log n})$ due to the recurrence equation $T(1) = 1$ and $T(l) = 2n \cdot T(l/2)$ resulting in $T(n) = (2n)^{\log n} = n^{1+\log n}$ time for one test. Varying $b$ and iterating on $l$ in the range of $\{1, \ldots, n\}$ gives the overall performance of $O(n^{3+\log n})$ steps.

```
Divide-And-Conquer-BFS(s)
  for i ← 1 to n
    for l ← 1 to n
      if (Path(s, i, l))
        print (s, i, l)
        break

Path(a, b, l)
  if ((a, b) ∈ E)
    return true
  else
    for j ← 1 to n
      if (Path(a, j, ⌈l/2⌉) and Path(j, b, ⌊l/2⌋))
        return true
    return false
```

Table 2.2: Computing the BFS Level.

### 2.2.2 Divide-And-Conquer SSSP

To extend this idea to the single-source shortest path problem (cf. Figure 2.3) with edge weights bounded by a constant $C$, we check the weights

| | |
|---|---|
| $\lfloor w/2 \rfloor - \lceil C/2 \rceil$ for path 1, | $\lfloor w/2 \rfloor + \lceil C/2 \rceil$ for path 2, |
| $\lfloor w/2 \rfloor - \lceil C/2 + 1 \rceil$ for path 1, | $\lfloor w/2 \rfloor + \lceil C/2 \rceil - 1$ for path 2, |
| ... | ... |
| $\lfloor w/2 \rfloor + \lceil C/2 \rceil$ for path 1, | $\lfloor w/2 \rfloor - \lceil C/2 \rceil$ for path 2. |

If there is a path with total weight $w$ then it can be decomposed into one of above partitions. The worst-case reduction on weights is $Cn \rightarrow Cn/2 + C/2 \rightarrow Cn/4 + 3C/4 \rightarrow \ldots \rightarrow C \rightarrow C - 1 \rightarrow C - 2 \rightarrow C - 3 \rightarrow \ldots \rightarrow 1$.

```
Divide-And-Conquer-SSSP(s)
  for i ← 1 to n
    for w ← 1 to C · n
      if (Path(s, i, w))
        print (s, i, w)
        break

Path(a, b, w)
  if (weight(a, b) = w)
    return true
  else
    for j ← 1 to n
      for s ← max{1, ⌊w/2⌋ − ⌈C/2⌉}
            to min{w − 1, ⌈w/2⌉ + ⌈C/2⌉}
        if (Path(a, j, s) and Path(j, b, w − s))
          return true
    return false
```

Table 2.3: Searching the Shortest Paths.

Therefore, the recursion depth is bounded by $\log(Cn) + C$ which results in a space requirement of $O(\log n)$ integers. As in the BFS case this compares to exponential time.

We do not claim practical applicability of these algorithms but want to make a start towards efficient shortest path algorithms for relatively little memory and unmodifiable large data, for example on optical read-only storage. In particular, time-space trade-offs seem to require new techniques.

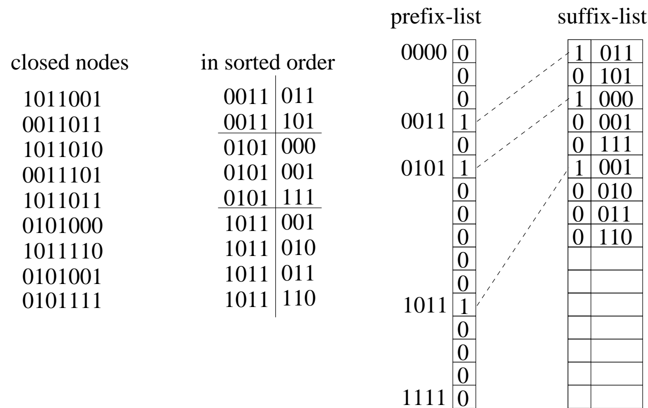


Figure 2.1: Example for Suffix Lists with $p = 4$, and $s = 3$.

## 2.3 Suffix Lists

Given $m$ bits of memory, we want to maintain a dynamically evolving visited list *closed* under inserts and membership queries. The entries of *closed* are integers from $\{0, n\}$. Let $r$ denote the maximal size of closed nodes that can be accommodated. As long as $n \leq m$ a simple bit array with bit $i$ denoting element $i$ is sufficient. Using hashing with open addressing, $r$ is limited to $O(n/\log n)$. In the following we describe a simple but very space efficient approach with small update and query times. Similar ideas appeared in [59] but the data structure there is static and not theoretically analyzed. Another dynamic variant achieving asympotically equivalent storage bounds as our approach is sketched in [47]. However, constants are only given for two static examples. We provide constants for the dynamic version; comparing with the numbers of [47], our dynamic version could host up to five times more elements of the same value range. However, one has to take into consideration that the data structure of [47] provides constant access time whereas our structure incurs amortized logarithmic access time.

### 2.3.1 Representation

Let *bin*$(u)$ be the binary representation of an element $u \leq n$ from the set *closed*. We split *bin*$(u)$ in $p$ high bits and $s = \lceil \log n \rceil - p$ low bits. Furthermore, $u_{s+p-1}, \ldots, u_s$ denotes the prefix of *bin*$(u)$ and $u_{s-1}, \ldots, u_0$ stands for the suffix of *bin*$(u)$.

A *suffix list data structure* consists of a linear array $P$ of size $2^p$ bits and of a two-dimensional array $L$ of size $r(m + 1)$ bits. The basic idea of suffix lists is to store a common prefix of several entries as a single bit in $P$, whereas the distinctive suffixes form a group within $L$. $P$ is stored as a bit array. $L$ can hold several groups with each group consisting of a multiple of $s + 1$ bits. The first bit of each $s + 1$-bit row in $L$ serves as a *group bit*. The first $s$ bit suffix entry of a group has group bit one, the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order, see Figure 2.1.

### 2.3.2 Searching

First, we compute $k = \sum_{i=0}^{p-1} u_{s+i} \cdot 2^i$ which gives us the search position in the prefix array $P$. Then we simply count the number of ones in $P$ starting from position $P[0]$ until we reach $P[k]$. Let $z$ be this number. Finally we search through $L$ until we have found the $z$th suffix of $L$ with group bit one. If we have to perform a membership query we simply search in this group. Note that searching a single entry may require scanning large areas of main memory.

### 2.3.3 Inserting

To insert entry $u$ we first search the corresponding group as described above. In case $u$ opens a new group within $L$ this involves setting group bits in $P$ and $L$. The suffix of $u$ is inserted in its group while maintaining the elements of the group sorted. Note that an insert may need to shift many rows in $L$ in order to create space at the desired position. The maximum number $r$ of elements that can be stored in $S$ bits is limited as follows: We need $2^p$ bits for $P$ and $s+1 = \lceil \log n \rceil - p + 1$ bits for each entry of $L$. Hence, we choose $p$ so that $r$ is maximal subject to

$$r \leq \frac{m - 2^p}{\lceil \log n \rceil - p + 1}.$$

For $p = \Theta(\log m - \log\log(n/m))$ the space requirement for both $P$ and the suffixes in $L$ is small enough to guarantee $r = \Theta\left(\frac{m}{\log(n/m)}\right)$.

### 2.3.4 Checkpoints

We now show how to speed up the operations. When searching or inserting an element $u$ we have to compute $z$ in order to find the correct group in $L$. Instead of scanning potentially large parts of $P$ and $L$ for each single query we maintain checkpoints, *one-counters*, in order to store the number of ones seen so far. Checkpoints are to lie close enough to support rapid search but must not consume more than a small fraction of the main memory. For $2^p \leq r$ we have $z \leq r$ for both arrays, so $\lceil \log r \rceil$ bits are sufficient for each one-counter.

Keeping one-counters after every $1/(c_1 \cdot \lfloor \log r \rfloor)$ entries limits the total space requirement. Binary search on the one-counters of $P$ now reduces the scan-area to compute the correct value of $z$ to $c_1 \cdot \lfloor \log r \rfloor$ bits.

Searching in $L$ is slightly more difficult because groups could extend over $2^s$ entries, thus potentially spanning several one-counters with equal values. Nevertheless, finding the beginning and the end of large groups is possible within the stated bounds. As we keep the elements within a group sorted, another binary search on the actual entries is sufficient to locate the position in $L$.

### 2.3.5 Buffers

We now turn to insertions where two problems remain: adding a new element to a group may need shifting large amount of data. Also, after each insert the checkpoints must be updated. A simple solution uses a second buffer data structure *BU* which is less space

efficient but supports rapid inserts and look-ups. When the number of elements in *BU* exceeds a certain threshold, *BU* is merged with the old suffix lists to obtain a new up-to-date space efficient representation. Choosing an appropriate size of *BU*, amortized analysis shows improved computational bounds for inserts while achieving asymptotically the same order of phases for the graph search algorithm.

Note that membership queries must be extended to *BU* as well. We implement *BU* as an array for hashing with open addressing. *BU* stores at most $c_2 \cdot r/\lceil \log n \rceil$ elements of size $p + s = \lceil \log n \rceil$, for some small constant $c_2$. As long as there is 10% space left in *BU*, we continue to insert elements into *BU* otherwise *BU* is sorted and the suffixes are moved from *BU* into the proper groups of $L$. The reason not to exploit the full hash table size is again to bound the expected search and insert time within *BU* to a constant number of tests.

**Theorem 1** *Searching and inserting $n$ items into suffix lists under space restriction $m$ can be done in $O(n \cdot \log^2 n)$ bit operations. Assuming $\log n$ bits for a machine word, the total run time for $n$ inserts and memberships is $O(n \log n)$.*

**Proof:** For a membership query we perform binary searches on numbers of $\lceil \log r \rceil$ bits or $s$ bits, respectively. So, to search an element we need $O(\log^2 r + s^2) = O(\log^2 n)$ bit operations since $r \leq n$ and $s \leq \log n$.

Each of the $O(r/\log n)$ buffer entries consists of $O(\log n)$ bits, hence sorting the buffer can be done with

$$O\left(\log n \cdot \frac{r}{\log n} \cdot \log \frac{r}{\log n}\right) = O(r \cdot \log n)$$

bit operations. Starting with the biggest occurring keys merging can be performed in $O(1)$ memory scans, $O(m)$ operations. This also includes updating all one-counters. In spite of the additional data structures we still have

$$r = \Theta\left(\frac{m}{\log(n/m)}\right).$$

Thus, the total bit complexity for $n$ inserts and membership queries is given by

$$\begin{aligned}
&O(\#\textit{buffer-runs}\ (\#\textit{sorting-ops} + \#\textit{merging-ops}) + \\
&\quad \#\textit{elements}\ \#\textit{buffer-search-ops} + \\
&\quad \#\textit{elements}\ \#\textit{membership-query-ops}) = \\
&O(n/r \cdot \log n \cdot (r \cdot \log n + m) + n \cdot \log^2 n + n \cdot \log^2 n) = \\
&O(n/r \cdot \log n \cdot (r \cdot \log n + r \cdot \log(n/m)) + n \cdot \log^2 n) = \\
&O(n \cdot \log^2 n).
\end{aligned}$$

Assuming a machine word length of $\log n$ in the RAM model, any modification or comparison of entries with $O(\log n)$ bits appearing in our suffix lists can be done using $O(1)$ machine operations. Hence the total complexity reduces to $O(n \cdot \log n)$ operations.

■

The constants can be improved using the following observation: in the case $n = (1 + \epsilon) \cdot m$, for a small $\epsilon > 0$ nearly half of the entries in $P$ will always be zero, namely those which are lexicographically bigger than the suffix of $n$ itself. Cutting the $P$ array at this position leaves more room for $L$ which in turn enables us to keep more elements.

### 2.3.6 The Information Theoretic Bound

We place an upper bound on the maximal size $r^*$ of the subset that can be stored. For the static case, we observe that $\lceil \log \binom{n}{r^*} \rceil \leq m$. However, if we consider the dynamic case, i.e. including insertions, we have to represent all former configurations. This results in

$$\left\lceil \log \left( \sum_{i=0}^{r^*} \binom{n}{i} \right) \right\rceil \leq m.$$

We aim choose $r^*$ maximal subject to this inequality. For $r^* \leq (n-2)/3$ we have

$$\binom{n}{r^*} \leq \sum_{i=0}^{r^*} \binom{n}{i} \leq 2 \cdot \binom{n}{r^*}.$$

The correctness follows from $\binom{n}{i} / \binom{n}{i+1} \leq 1/2$ for $i \leq (n-2)/3$. We are only interested in the logarithms, so we conclude

$$\log \binom{n}{r^*} \leq \log \left( \sum_{i=0}^{r^*} \binom{n}{i} \right) \leq \log \left( 2 \binom{n}{r^*} \right) = \log \binom{n}{r^*} + 1$$

Obviously in this restricted range it is sufficient to concentrate on the last binomial coefficient. The error in our estimate is at most one bit. The restriction on $r^*$ is compatible with all reasonable choices for $n$ and $m$. Using

$$\begin{aligned} \log \binom{n}{r^*} &= \log \frac{n \cdot (n-1) \cdot \ \cdots \ \cdot (n - r^* + 1)}{r^*!} \\ &= \sum_{j=n-r^*+1}^{n} \log j - \sum_{j=1}^{r^*} \log j, \end{aligned}$$

we can approximate the logarithm by two corresponding integrals. If we properly bias the integral limits we can be sure to compute a lower bound

$$\log \binom{n}{r^*} \geq \int_{n-r^*+1}^{n} \log(x) \, \mathrm{d}x - \int_{2}^{r^*+1} \log(x) \, \mathrm{d}x.$$

Maximizing $r^*$ with respect to this equation yields an information theoretic upper bound.

Table 2.4 compares suffix lists with hashing and open addressing. The constants for suffix lists are chosen so that $2 \cdot c_1 + c_2 \leq 1/10$ which means that if $r$ elements can be treated, we set aside $r/10$ bits to speed-up internal computations. For hashing with open addressing we also leave $10\%$ memory free to keep the internal computation time moderate. When using suffix lists instead of hashing, note that only the ratio between $n$ and $m$ is important. For the static data structure of [47] the following numbers are given: for $\frac{n}{m} = \frac{1.0 \cdot 2^{32}}{1.9 \cdot 2^{30}} \approx 1.05$ it can store a fraction of $\frac{r}{n} = \frac{1.4 \cdot 2^{27}}{1.0 \cdot 2^{32}} \approx 4.37\%$ of $n$. Our approach achieves $22.7\%$ which constitues an improvement by a factor of more than five. For another example with $n/m \approx 3.2$ our approach gains by a factor of about $1.8$.

Hence, suffix lists can close the phase gap in search algorithms between the upper bound and trivial approaches like hashing with open addressing. Already for $n \geq 1.1 \cdot m$ we reach two-optimality.

| $n/m$ | Upper Bound | Suffix Lists | Hashing $n = 2^{20}$ | Hashing $n = 2^{30}$ |
|---|---|---|---|---|
| 1.05 | 33.2 % | 22.7 % | 4.3 % | 2.9 % |
| 1.10 | 32.4 % | 21.2 % | 4.1 % | 2.8 % |
| 1.25 | 24.3 % | 17.7 % | 3.6 % | 2.4 % |
| 1.50 | 17.4 % | 13.4 % | 3.0 % | 2.0 % |
| 2.00 | 11.0 % | 9.1 % | 2.3 % | 1.5 % |
| 3.00 | 6.1 % | 5.3 % | 1.5 % | 1.0 % |
| 4.00 | 4.1 % | 3.7 % | 1.1 % | 0.7 % |
| 8.00 | 1.7 % | 1.5 % | 0.5 % | 0.4 % |
| 16.00 | 0.7 % | 0.7 % | 0.3 % | 0.2 % |

Table 2.4: Fractions of $n$ stored in Suffix Lists and Hashing with Open Addressing.

## 2.4 Bit-State Hash-Tables

Advanced to the treatment of data structures and algorithms we give a small introduction to the verification of distributed software systems and communication protocols; an apparent and practical relevant domain for state-space search.

### 2.4.1 State Space Search for Protocols Validation

Reliable communication is probably the most important issue for accessing the Internet and for the design of distributed computer systems. Usually a layered structure like the *ISO Reference Model* is used to allow for different abstractions. In one layer (transport layer) we have the request for reliable communication while the next lower layers provide this quality of service facing a lossy channel (cf. Figure 2.2).

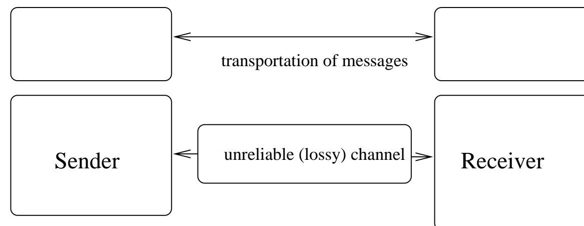


Figure 2.2: Communication over a Lossy Channel for Messaging in Layered Protocols.

One example to cope with lossy channels is the alternating bit protocol. The message flow is visualized in Figure 2.3. To assert secure data transport from the sender to the receiver we assume sequence numbers for messages. In the following we study algorithms and data structures to certify the correctness of a such a protocol.

### 2.4.2 Supertrace

The idea of bit-state hashing is adopted from Holzman's protocol validator Spin [189], that parses the expressive concurrent Promela protocol specification language. It com-

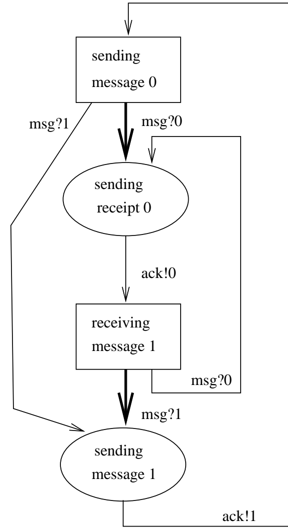


Figure 2.3: Flow of Control on a Lossy Channel with the Alternating Bit Protocol.

presses the state description of several hundred bits down to only a few bits to build a hash table with up to $2^{30}$ entries and more. Combined with a depth-first search strategy this is in fact the *supertrace algorithm*: A state $s$ is represented by its hash address $h(s)$. When generating a state the corresponding bit is set. Synonyms are regarded as duplicates resulting in pruning the search. The search algorithm is not complete, since not all synonyms are disambiguated. Moreover, through depth-first traversal, the length of a witness for an encountered error state is not minimal.

### 2.4.3 Data Structures

As an illustration and generalization of the bit-state hashing idea, Figure 2.4 depicts the range of possible hash structures: Usual hashing with chaining of synonyms, single-bit hashing, double-bit hashing and hash compact [331]. Let $n$ be the number of reachable states and $m$ be the maximal number of bits available. A coarse approximation for single bit-state hashing coverage with $n < m$ is $1 - P_1$ with the average probability of collision $P_1 \leq \frac{1}{n}\sum_{i=0}^{n-1}\frac{i}{m} \leq n/2m$, since the $i$-th element collides with one of the $i-1$ already inserted elements with a probability of at most $(i-1)/m$, $1 \leq i \leq n$ [189]. For multi-bit hashing and $h$ (independent) hash-functions by assuming $hn < m$ coverage is improved to $1 - P_h$ with average probability of collision $P_h \leq \frac{1}{n}\sum_{i=0}^{n-1}(h \cdot \frac{i}{m})^h$, since $i$ elements occupy at most $hi/m$ addresses, $0 \leq i \leq n-1$. For double bit-state hashing this simplifies to $P_2 \leq \frac{1}{n}(\frac{2}{m})^2\sum_{i=0}^{n-1} i^2 = 2(n-1)(2n-1)/3m^2 \leq 4n^2/3m^2$.

### 2.4.4 Sequential and Universal Hashing

The drawback in incompleteness of partial search is compensated by re-invoking the algorithm with different hash functions to improve the coverage of the search tree. Subsequently, this technique, called *sequential hashing*, examines various beams in the search tree (up to a certain threshold depth). In considerably large protocols supertrace with sequential hashing succeeds in finding bugs but still returns long witness trails. If in se-

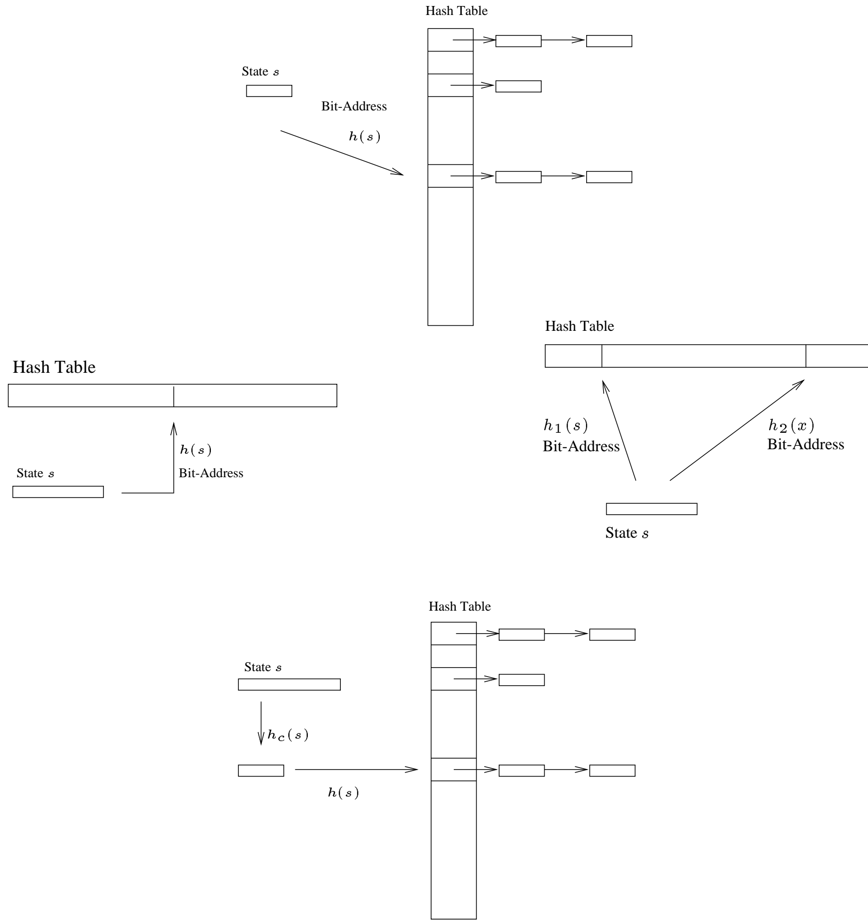


Figure 2.4: Ordinary Hashing, Single Bit-State Hashing, Double Bit-State Hashing, and Hash-Compact.

quential hashing exploration with the first hash first function covers $m/n$ of the search space, the probability that a state $x$ is not generated in $d$ independent runs is $(1 - m/n)^d$ such that $x$ is reached with probability $1-(1-m/n)^d$. Eckerle and Lais [89] have shown that this *ideal* circumstances are not given in practice and refine the model for coverage prediction.

Moreover, universal hash functions suit best for implementing sequential hashing. Let $A$, $B$ be sets with $|B| = 2^w$, for some integer value $w$. The class of hash functions $\mathcal{H}$ is *universal*, if for all $x$, $y \in A$, we have

$$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{|B|}.$$

Universal hash functions lead to a good distribution of values on the average. If $h$ is drawn randomly from $\mathcal{H}$ and $S$ is the set of keys to be inserted in the hash table, the expected cost of each search, insert and delete operation is bounded by $(1 + |S|/|B|)$. We give an example of a universal hash function. Let $p$ be prime, and $p \geq |A|$ and $h_{m,n}(s) = ((m \cdot s + n) \bmod p) \bmod |B|$. Then the class $\mathcal{H}_1 := \{h_{m,n} | m, n \in Z_p\}$ is universal.

```
Partial IDA*(s)
    Push(S, s, h(s)); U' ← U ← h(s)
    while (U ≠ ∞)
        U ← U'; U' ← ∞
        Init(H)
        while (S ≠ ∅)
            (u, f(u)) ← Pop(S)
            if (goal(u)) return (u, f(u))
            for all v in Γ(u)
                if (Search(H, v) ≠ ∅)
                    Insert(H, v)
                    if (f(u) + w(u, v) − h(u) + h(v) > U)
                        if (f(u) + w(u, v) − h(u) + h(v) < U')
                            U' ← f(u) + w(u, v) − h(u) + h(v)
                    else
                        Push(S, v, f(u) + w(u, v) − h(u) + h(v))
```

Table 2.5: Partial IDA* Algorithm.

### 2.4.5 Validating Process

For the validation of the design of the protocols, bug-finding by simulation and testing has its drawbacks, since several subtle bugs in concurrent systems are difficult to establish. Given a formal specification of a desired protocol property model-checking is a push-button procedure to verify the correctness. Validation is performed by traversing the finite-state machine representation of the protocol to find a bug. Therefore protocols are represented by state spaces, in which reachability analysis is performed to establish error states.

Therefore, directed search for minimal counterexamples in the protocol space according to a given implementation corresponds to the search for an optimal solution with the goal as the failure state. From a model checking perspective [65] the approach allows to implement various heuristics to direct the search into the direction of the failure. From an AI-perspective partial search, maybe assisted with sequential hashing, condenses duplicate information in various search and planning problem spaces.

### 2.4.6 Heuristic Search Algorithm

The apparent aspirant for state compaction is IDA* with *transposition tables*, since, in opposite to A*, it tracks the solution path on the stack, which allows to omit the predecessor link in the state description of the set of visited states.

When substituting the transposition table $H$ of already visited nodes in IDA* by bit-state, multi bit-state or hash compaction we establish the *Partial IDA** algorithm as depicted in Table 2.5. Since neither the predecessor nor the $f$-value are present, in order to distinguish the current iteration from the previous ones, the bit-state table has to be re-initialized in each iteration of IDA*. Refreshing large bit-vector tables is fast in practice,

but for shallow searches with a small number of expanded nodes this scheme can be improved by invoking ordinary IDA* with transposition table updates for smaller thresholds and by applying bit-vector exploration in large depths only.

In practice the obtained counterexamples are minimal, since the coverage with bit-state duplicate elimination is very close to 100 % for moderately sized systems ($n < m$). Moreover, the technique of *trail-directed search* can effectivly improve non-optimal existing paths [110].

The results for searching deadlocks in one large communication protocol are depicted in Table 2.6, where the number of expansions with respect to different optimal search algorithms for an increasing threshold is shown. For A* a snapshot is taken at each time the priority queue value increases, while in IDA* the number of expanded nodes according to each completed iteration is shown. Hence, the number of node expansion numbers in these two algorithms do not match exactly, but indicate a common trend. The considered protocol instance is the industrial General Inter-ORB Protocol (*GIOP*, 1 server and 3 clients) [201], which is a key component of the Common Object Request Broker Architecture (*CORBA*) specification.

The witness for a seeded deadlock in depth 70 has to be established according to the heuristic that counts the number of non-active processes. The state vector generated by the validator tool SPIN is 544 Bytes large, such that the visited list (hash table or transposition table) is bounded to $2^{18}$ states corresponding to approx. $2^{17}$ KByte or $128$ MByte. Therefore, we fix the size of the bit-state hash table accordingly at $2^{30}$ Bits.

Algorithm A* exceeds its space limit in depth 61 and fails. IDA* utilizes a transposition table which is exhausted at the same depth. As IDA* then searches the tree of generation paths it compensates space for time. But even when investing more than 24 hours on our 248 MHz Sun Ultra Workstation and when utilizing the table constructed so far, ordinary IDA* was not able to complete search depth 61. On the other hand, Partial A* finishes all searches up to depth 70 with either single- and double bit-state hashing within a total of one hour.

Since the algorithms are not complete, we validated optimality with A* with our maximum of 1.5 GByte main memory. Note that the difference in the number of node expansions in single and double bit-state hashing is very small (less than a hundred) and only occurs in large search depths (iteration 58 onwards). As Partial IDA* with double bit-state hashing expands exactly the same number of states as IDA* with a transposition table, we actually observe no loss of information in the example.

## 2.5 Conclusion

At the limit of main memory eliminating duplicates and weight diversity can soon result in thrashing both resources time and space, such that powerful data structures for caching, partial search and compressed dictionaries are required. Therefore, regarding the limits and possibilities of A*, we have suggested different contributions to memory-restricted search. Partial search supports bookkeeping in tremendously large hash tables to avoid duplicates in the search, while suffix lists push the envelope for increasing the number of nodes to be stored without loss of information.

The treatment of Partial IDA* search elaborates on precursoring findings in [109], where a rudimentory bit vector and single-bit hashing function has been chosen for im-

plementation. For the experiments we chose a non-trivial protocol example [107], but recent progress shows that the algorithm has also reduced the search efforts for optimally solving Atomix, a PSPACE-complete AI single-agent search problem [194]. Omitting the visited list and exploring the space in a Divide-and-Conquer fashion has been proposed in [217], and the algorithms we consider study the effect of removing the horizon-list as well. Another model checking approach for state compression as to answer to the representation problem of large sets of states are binary decision diagrams (BDDs) that are able to encode large sets of states without necessarily encountering exponential growth. However, hybrid methods of explicite and symbolic search methods are still to be developed.

| depth | A* (hash table) | IDA* (transposition table) | Partial IDA* (single bit-state) | Partial IDA* (double bit-state) |
|---|---|---|---|---|
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 40 | 6,646 | 6,333 | 6,333 | 6,333 |
| 41 | 9,306 | 8,184 | 8,184 | 8,184 |
| 42 | 10,955 | 10,575 | 10,575 | 10,575 |
| 43 | 13,666 | 13,290 | 13,290 | 13,290 |
| 44 | 17,761 | 16,500 | 16,500 | 16,500 |
| 45 | 20,130 | 19,860 | 19,860 | 19,860 |
| 46 | 25,426 | 23,646 | 23,646 | 23,646 |
| 47 | 27,714 | 27,654 | 27,654 | 27,654 |
| 48 | 33,799 | 32,040 | 32,040 | 32,040 |
| 49 | 37,095 | 37,011 | 37,011 | 37,011 |
| 50 | 46,105 | 42,849 | 42,849 | 42,849 |
| 51 | 51,113 | 49,872 | 49,872 | 49,872 |
| 52 | 61,710 | 58,545 | 58,545 | 58,545 |
| 53 | 73,195 | 69,162 | 69,162 | 69,162 |
| 54 | 85,245 | 81,993 | 81,993 | 81,993 |
| 55 | 96,995 | 96,543 | 96,543 | 96,543 |
| 56 | 113,950 | 112,296 | 112,296 | 112,296 |
| 57 | 115,460 | 129,138 | 129,138 | 129,138 |
| 58 | 147,042 | 146,625 | 146,623 | 146,625 |
| 59 | 150,344 | 164,982 | 164,978 | 164,982 |
| 60 | 184,872 | 184,383 | 184,376 | 184,383 |
| 61 | 187,411 | 206,145 | 206,135 | 206,145 |
| 62 | - | $> 97{,}157{,}721$ | 229,611 | 229,626 |
| 63 | - | - | 255,386 | 255,411 |
| 64 | - | - | 282,416 | 282,444 |
| 65 | - | - | 311,306 | 311,340 |
| 66 | - | - | 341,522 | 341,562 |
| 67 | - | - | 373,374 | 373,422 |
| 68 | - | - | 407,249 | 407,310 |
| 69 | - | - | 442,863 | 442,941 |
| 70 | - | - | 67 | 67 |

Table 2.6: Number of Expanded nodes of Search Algorithms in the GIOP Protocol.

# Paper 3

# Time Complexity of Iterative-Deepening-A*

Richard E. Korf.
Computer Science Department
University of California
Los Angeles, Ca. 90095
eMail: korf@cs.ucla.edu

Michael Reid.
Department of Mathematics and Statistics
University of Massachusetts
Amherst, MA 01003-4515
eMail: reid@math.umass.edu

Stefan Edelkamp.
Institut für Informatik
Georges-Köhler-Allee, Gebäude 51
79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

*Artificial Intelligence*, Vol 129, No. 1-2, pages 199-218, 2001.

**Abstract**

We analyze the time complexity of iterative-deepening-A* (IDA*). We first show how to calculate the exact number of nodes at a given depth of a regular search tree, and the asymptotic brute-force branching factor. We then use this result to analyze IDA* with a consistent, admissible heuristic function. Previous analyses relied on an abstract analytic model, and characterized the heuristic function in terms of its accuracy, but do not apply to concrete problems. In contrast, our analysis allows us to accurately predict the performance of IDA* on actual problems such as the sliding-tile puzzles and Rubik's Cube. The heuristic function is characterized by the distribution of heuristic values over the problem space. Contrary to conventional wisdom, our analysis shows that the asymptotic heuristic branching factor is the same as the brute-force branching factor. Thus, the effect of a heuristic function is to reduce the effective depth of search by a constant, relative to a brute-force search, rather than reducing the effective branching factor.

## 3.1 Introduction and overview

Our goal is to predict the running time of iterative-deepening-A* (IDA*) [213], a linear-space version of the A* algorithm [161]. Both these algorithms rely on a heuristic evaluation function $h(n)$ that estimates the cost of reaching a goal from node $n$. If $h(n)$ is admissible, or never overestimates actual cost from node $n$ to a goal, then both algorithms return optimal solutions.

The running time of IDA* is usually proportional to the number of nodes expanded. This depends on the cost of an optimal solution, the number of nodes in the brute-force search tree, and the heuristic function. In Section 3.2, we show how to compute the size of a brute-force search tree, and its asymptotic branching factor. In Section 3.3, we use this result to predict the number of nodes expanded by IDA* using a consistent heuristic function. The key to this analysis is characterizing the heuristic function.

Previous work on this problem characterized the heuristic by its accuracy as an estimate of actual solution cost. The accuracy of a heuristic is very difficult to obtain, and the corresponding asymptotic results, based on an abstract model, don't predict performance on concrete problems. In contrast, we characterize a heuristic by its distribution of values, a characterization that is easy to determine. As a result, we can predict the performance of IDA* on the sliding-tile puzzles and Rubik's Cube to within 1% of experimental results. In contrast to previous work, our analysis shows that the asymptotic heuristic branching factor is the same as the brute-force branching factor. This implies that the effect of a heuristic function is to reduce the effective depth of search by a constant, relative to a brute-force search, rather than reducing the effective branching factor.

Much of this work originally appeared in two AAAI-98 papers, one on the brute-force branching factor [104], and the other on the analysis of IDA* [219]. We begin with brute-force search trees.

## 3.2 Branching factor of regular search trees

### 3.2.1 Graph versus tree-structured problem spaces

Most problem spaces are graphs with cycles. Given a root node of any graph, however, it can be expanded into a tree. For example, Fig. 3.1 shows a search graph, and the top part of its tree expansion, rooted at node A. In a tree expansion of a graph, each distinct path to a node of the graph generates a different node of the tree. The tree expansion of a graph can be much larger than the original graph, and in fact is often infinite even for a finite graph.

In this paper, we focus on problem-space trees. The reason is that IDA* uses depth-first search to save memory, and hence cannot detect most duplicate nodes. Thus, it potentially explores every path to a given node, and searches the tree-expansion of the problem-space graph. We can characterize the size of a brute-force search tree by its asymptotic branching factor. The branching factor of a node is the number of children it has. In most trees, however, different nodes have different numbers of children. In that case, we define the *asymptotic branching factor* as the number of nodes at a given depth,

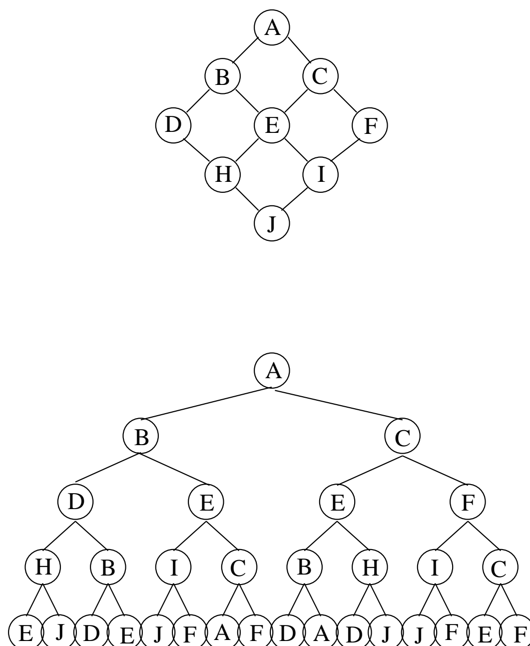


Figure 3.1: Graph and part of its tree expansion.

divided by the number of nodes at the next shallower depth, in the limit as the depth goes to infinity.

We present examples of problem-space trees, and compute their asymptotic branching factors. We formalize the problem as the solution of a set of simultaneous equations. We present both analytic and numerical techniques for computing the exact number of nodes at a given depth, and determining the asymptotic branching factor. We give the branching factors of Rubik s Cube and sliding-tile puzzles from the Five Puzzle to the Ninety-Nine Puzzle.

### 3.2.2 Example: Rubik's Cube

Consider Rubik's Cube, shown in Fig. 3.2. We define any 90, 180, or 270 degree twist of a face as one move. Since there are six faces, this gives an initial branching factor of $6 \cdot 3 = 18$. We never twist the same face twice in a row, however, since the same result can be obtained with a single twist of that face. This reduces the branching factor to $5 \cdot 3 = 15$ after the first move.

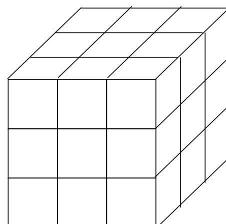

Figure 3.2: Rubik's Cube.

Note that twists of opposite faces are independent of each other and hence commute. For example, twisting the left face followed by the right face gives the same result as

twisting the right face followed by the left face. Thus, if two opposite faces are twisted consecutively, we require them to be twisted in a particular order, to eliminate the same state resulting from twisting them in the opposite order. For each pair of opposite faces, we arbitrarily label one a “first” face, and the other a “second” face. Thus, if Left, Up and Front were the first faces, then Right, Down, and Back would be the second faces. After a first face is twisted, there are three possible twists of each of the remaining five faces, for a branching factor of 15. After a second face is twisted, however, we can only twist four remaining faces, excluding the face just twisted and its corresponding first face, for a branching factor of 12. Thus, the asymptotic branching factor is between 12 and 15. The exact asymptotic branching factor depends on the relative fraction of nodes where the last move was a twist of a first face (type-1 nodes), or a twist of a second face (type-2 nodes). Define the *equilibrium fraction* of type-1 nodes as the number of type-1 nodes at a given depth, divided by the total number of nodes at that depth, in the limit of large depth. The equilibrium fraction is not $1/2$, because a twist of a first face can be followed by a twist of any second face, but a twist of a second face cannot be followed immediately by a twist of the corresponding first face. To determine the asymptotic branching factor, we need the equilibrium fraction of type-1 nodes. The fraction of type-2 nodes is one minus the fraction of type-1 nodes. Each type-1 node generates $2 \cdot 3 = 6$ type-1 nodes and $3 \cdot 3 = 9$ type-2 nodes as children, the difference being that you can't twist the same first face again. Each type-2 node generates $2 \cdot 3 = 6$ type-1 nodes and $2 \cdot 3 = 6$ type-2 nodes, since you can't twist the corresponding first face next, or the same second face again. Thus, the number of type-1 nodes at a given depth is 6 times the number of type-1 nodes at the previous depth, plus 6 times the number of type-2 nodes at the previous depth. The number of type-2 nodes at a given depth is 9 times the number of type-1 nodes at the previous depth, plus 6 times the number of type-2 nodes at the previous depth.

Let $f_1$ be the fraction of type-1 nodes, and $f_2 = 1 - f_1$ the fraction of type-2 nodes at a given depth. If $n$ is the total number of nodes at that depth, then there will be $nf_1$ type-1 nodes and $nf_2$ type-2 nodes at that depth. In the limit of large depth, the fraction of type-1 nodes will converge to the equilibrium fraction, and remain constant. Thus, at large depth,

$$
\begin{aligned}
f_1 &= \frac{\text{type-1 nodes at next level}}{\text{total nodes at next level}} = \frac{6nf_1 + 6nf_2}{6nf_1 + 6nf_2 + 9nf_1 + 6nf_2} \\
&= \frac{6f_1 + 6(1 - f_1)}{15f_1 + 12(1 - f_1)} = \frac{6f_1 + 6f_2}{15f_1 + 12f_2} = \frac{6}{3f_1 + 12} = \frac{2}{f_1 + 4} = f_1
\end{aligned}
$$

Cross multiplying gives us the quadratic equation $f_1^2 + 4f_1 = 2$, which has a positive root at $f_1 = \sqrt{6} - 2 \approx 0.44949$. This gives us an asymptotic branching factor of $15 \cdot f_1 + 12 \cdot (1 - f_1) = 3\sqrt{6} + 6 \approx 13.34847$.

### 3.2.3 A system of simultaneous equations

In general, this analysis produces a system of simultaneous equations. For another example, consider the Five Puzzle, the $2 \times 3$ version of the well-known sliding-tile puzzles (see Fig. 3.3A).

In this problem, the branching factor of a node depends on the blank position. In Fig. 3.3B, the positions are labelled $s$ and $c$, representing side and corner positions, re-

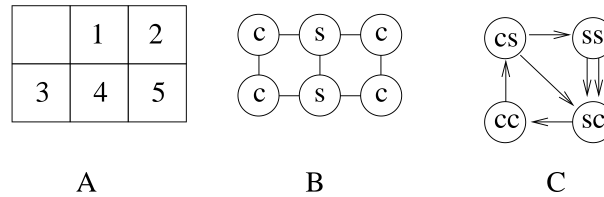


Figure 3.3: The Five Puzzle.

spectively. We don't generate the parent of a node as one of its children, to avoid duplicate nodes representing the same state. This requires keeping track of both the current and previous blank positions. Let $cs$ denote a node where the blank is currently in a side position, and the last blank position was a corner position. Define $ss$, $sc$ and $cc$ nodes analogously. Since $cs$ and $ss$ nodes have two children each, and $sc$ and $cc$ nodes have only one child each, we have to know the equilibrium fractions of these different types of nodes to determine the asymptotic branching factor. Fig. 3.3C shows the different types of states, with arrows indicating the type of children they generate. For example, the double arrow from ss to sc indicates that each ss node generates two sc nodes at the next level.

Let $N(t, d)$ be the number of nodes of type $t$ at depth $d$ in the search tree. Then, we can write the following recurrence relations directly from the graph in Fig. 3.3C. For example, the last equation comes from the fact that there are two arrows from $ss$ to $sc$, and one arrow from $cs$ to $sc$.

$$\begin{aligned}
N(cc, d+1) &= N(sc, d), \\
N(cs, d+1) &= N(cc, d), \\
N(ss, d+1) &= N(cs, d), \\
N(sc, d+1) &= 2N(ss, d) + N(cs, d).
\end{aligned}$$

The initial conditions are that the first move either generates an ss node and two $sc$ nodes, or a $cs$ node and a $cc$ node, depending on whether the blank starts in a side or corner position, respectively.

**Numerical solution**

A simple way to compute the branching factor is to numerically compute the values of successive terms of these recurrences, until the relative frequencies of different state types converge. Let $f_{cc}$, $f_{cs}$, $f_{ss}$ and $f_{sc}$ be the number of nodes of each type at a given depth, divided by the total number of nodes at that depth. After a hundred iterations, we get the equilibrium fractions $f_{cc} = 0.274854$, $f_{cs} = 0203113$, $f_{ss} = 0.150097$, and $f_{sc} = 0.371936$. Since $cs$ and $ss$ states generate two children each, and the others generate one child each, the asymptotic branching factor is $f_{cc} + 2 \cdot f_{cs} + 2f_{ss} + f_{sc} = 1.35321$. Alternatively, we can simply compute the ratio between the total nodes at two successive depths to get the branching factor. The running time of this algorithm is the product of the number of different types of states, e.g., four in this case, and the search depth. In contrast, searching the actual tree to depth 100 would generate over $10^{13}$ states in this case.

**Analytical solution**

To compute the exact branching factor, we assume that the fractions eventually converge to constant values. This generates a set of equations, one from each recurrence. Let $b$ represent the asymptotic branching factor. If we view $f_{cc}$ as the number of $cc$ nodes at depth $d$, for example, then the number of $cc$ nodes at depth $d+1$ will be bf $cc$. This allows us to rewrite the above recurrences as the following set of equations. The last one constrains the fractions to sum to one.

$$\begin{aligned} bf_{cc} &= f_{sc} \\ bf_{cs} &= f_{cc} \\ bf_{ss} &= f_{cs} \\ bf_{sc} &= 2f_{ss} + f_{cs} \\ 1 &= f_{cc} + f_{cs} + f_{ss} + f_{sc} \end{aligned}$$

Repeated substitution to eliminate variables reduces this system of five equations in five unknowns to the single equation, $b^4 + b - 2 = 0$, with a solution of $b \approx 1.35321$. In general, the degree of the polynomial will be the number of different types of states. The Fifteen Puzzle, for example, has three types of positions, and six types of states.

If we make the naive and incorrect assumption that each blank position is equally likely in the Five Puzzle, we get an incorrect branching factor of $(2 \cdot 2 + 1 \cdot 4)/6 = 1.33333$. Another natural but erroneous approach is to include the parent of a node as one of its children, compute the resulting branching factor, and then subtract one from the result to eliminate the inverse of the last move. This gives an incorrect branching factor of $1.4142$ for the Five Puzzle. The error here is that eliminating the inverse of the last move changes the equilibrium fractions of the different types of states.

### 3.2.4 Results

We computed the asymptotic branching factors of square sliding-tile puzzles up to $10 \times 10$. Table 3.1 gives the even- and odd-depth branching factors for each puzzle. The last column is their geometric mean, or the square root of their product, which is the best estimate of the overall branching factor. Most of these values were computed by numerical iteration of the recurrence relations. As n goes to infinity, all the values converge to three, the branching factor of an infinite sliding-tile puzzle, since most positions have four neighbors, one of which was the previous blank position.

To see why the even and odd branching factors are different, color the positions of a puzzle in a checkerboard pattern, and note that the blank always moves between squares of different colors. If the sets of different-colored squares are equivalent to each other, as in the Five and Fifteen Puzzles, there is one branching factor. If the sets of different-colored squares are different however, as in the Eight Puzzle, there will be different even and odd branching factors. In general, an $n \times m$ sliding-tile puzzle will have different branching factors if and only if both $n$ and $m$ are odd.

| $n$ | $n^2-1$ | Even depth | Odd depth | Mean |
|---|---|---|---|---|
| 3 | 8 | 1.5 | 2 | $\sqrt{3}$ |
| 4 | 15 | 2.1304 | 2.1304 | 2.1304 |
| 5 | 24 | 2.30278 | 2.43426 | 2.36761 |
| 6 | 35 | 2.51964 | 2.51964 | 2.51964 |
| 7 | 48 | 2.59927 | 2.64649 | 2.62277 |
| 8 | 63 | 2.69590 | 2.69590 | 2.69590 |
| 9 | 80 | 2.73922 | 2.76008 | 2.74963 |
| 10 | 99 | 2.79026 | 2.79026 | 2.79026 |

Table 3.1: The asymptotic branching factor for the $(n^2-1)$-Puzzle

### 3.2.5 Generality of this technique

In some problem spaces, every node has the same branching factor. In other spaces, every node may have a different branching factor, requiring exhaustive search to compute the average branching factor. The technique described above determines the size of a brute-force search tree in intermediate cases, where there are a small number of different types of states, whose generation follows a regular pattern. Computing the size of the brute-force search tree is the first step in determining the time complexity of IDA*, our next topic.

## 3.3 Time complexity of IDA*

IDA* [213] uses the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the sum of the edge costs from the initial state to node $n$, and $h(n)$ is an estimate of the cost of reaching a goal from node $n$. Each iteration is a depth-first search where a branch is pruned when it reaches a node whose total cost exceeds the cost threshold of that iteration. The cost threshold for the first iteration is the heuristic value of the initial state, and increases in each iteration to the lowest cost of all nodes pruned on the previous iteration. It continues until a goal node is found whose cost does not exceed the current cost threshold.

### 3.3.1 Previous work

Most previous analyses of heuristic search focused on A* [138, 287, 295], and used an abstract problem-space tree where every node has $b$ children, every edge has unit cost, and there is a single goal node at depth $d$. The heuristic is characterized by its error in estimating actual solution cost. This model predicts that a heuristic with constant absolute error results in linear time complexity, while constant relative error results in exponential time complexity [138, 295]. There are several limitations of this model. The first is that it assumes there is only one path from the start to the goal state, whereas most problem spaces contain multiple paths to each state. The second limitation is that in order to determine the accuracy of the heuristic on even a single state, we have to determine the optimal solution cost from that state, which is expensive to compute. Doing this for a significant number of states is impractical for large problems. Finally, the results are

only asymptotic, and don't predict actual numbers of node generations. Because of these limitations, the previous work cannot be used to accurately predict the performance of A* or IDA* on concrete problems with real heuristics. That requires a different approach.

### 3.3.2 Overview

We begin with the consistency property of heuristics, and the conditions for node expansion by A* or IDA*. Next, we characterize a heuristic by the distribution of heuristic values over the problem space. Our main result is a formula for the number of node expansions as a function of the heuristic distribution, the cost threshold of an iteration, and the number of nodes of each cost in a brute-force search. Finally, we compare our analytic predictions with experimental data on Rubik s Cube and the Eight and Fifteen Puzzles. One implication of our analysis is that the effect of a heuristic function is to decrease the effective depth of search by a constant, rather than reducing the effective branching factor.

### 3.3.3 Consistent heuristics

One property of the heuristic required by our analysis is that it be *consistent*. A heuristic function $h(n)$ is consistent if for any node $n$ and any neighbor $n'$, $h(n) \leq k(n,n')+h(n')$, where $k(n,n')$ is the cost of the edge from $n$ to $n'$ [287]. An equivalent definition of consistency is that for any pair of nodes $n$ and $m$, $h(n) \leq k(n,m) + h(m)$, where $k(n,m)$ is the cost of an optimal path from $n$ to $m$. Consistency is similar to the triangle inequality of metrics, and implies admissibility, but not vice versa. However, most naturally occurring admissible heuristic functions are consistent as well [287].

### 3.3.4 Conditions for node expansion

We measure the time complexity of IDA* by the number of node expansions. If a node can be expanded and its children evaluated in constant time, the asymptotic time complexity of IDA* is simply the number of node expansions. Otherwise, it s the product of the number of node expansions and the time to expand a node. Given a consistent heuristic function, both A* and IDA* must expand all nodes whose total cost, $f(n) = g(n) + h(n)$, is less than $c$, the cost of an optimal solution [287]. Some nodes with the optimal solution cost may be expanded as well, until a goal node is chosen for expansion, and the algorithms terminate. In other words, $f(n) < c$ is a sufficient condition for A* or IDA* to expand node $n$, and $f(n) \leq c$ is a necessary condition. For a worst-case analysis, we adopt the weaker necessary condition.

An easy way to understand the node expansion condition is that any search algorithm that guarantees optimal solutions must continue to expand every possible solution path, until its cost is guaranteed to exceed the cost of an optimal solution, lest it lead to a better solution. On the final iteration of IDA*, the cost threshold will equal $c$, the cost of an optimal solution. In the worst case, IDA* will expand all nodes n whose cost $f(n) = g(n)+h(n) \leq c$. We will see below that this final iteration determines the overall asymptotic time complexity of IDA*.

### 3.3.5 Characterization of the heuristic

Previous analyses characterized the heuristic function by its accuracy as an estimator of optimal costs. As explained above, this is difficult to determine for a real heuristic, since obtaining optimal solutions is extremely expensive. In contrast, we characterize a heuristic function by the distribution of heuristic values over the nodes in the problem space. In other words, we need to know the number of states with heuristic value 0, how many states have heuristic value 1, the number with heuristic value 2, etc. Equivalently, we can specify this distribution by a set of parameters $D(h)$, which is the fraction of total states of the problem whose heuristic value is less than or equal to $h$. We refer to this set of values as the *overall distribution* of the heuristic. $D(h)$ can also be defined as the probability that a state chosen randomly and uniformly from all states in the problem has heuristic value less than or equal to $h$ . $h$ can range from zero to infinity, but for all values of h greater than or equal to the maximum value of the heuristic, $D(h) = 1$.

| h | States | Sum | $D(h)$ | Corner | Side | Csum | Ssum | $P(h)$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0.002778 | 1 | 0 | 1 | 0 | 0.002695 |
| 1 | 2 | 3 | 0.008333 | 1 | 1 | 2 | 1 | 0.008333 |
| 2 | 3 | 6 | 0.016667 | 1 | 2 | 3 | 3 | 0.016915 |
| 3 | 6 | 12 | 0.033333 | 5 | 1 | 8 | 4 | 0.033333 |
| 4 | 30 | 42 | 0.116667 | 25 | 5 | 33 | 9 | 0.115424 |
| 5 | 58 | 100 | 0.277778 | 38 | 20 | 71 | 29 | 0.276701 |
| 6 | 61 | 161 | 0.447222 | 38 | 23 | 109 | 52 | 0.446808 |
| 7 | 58 | 219 | 0.608333 | 41 | 17 | 150 | 69 | 0.607340 |
| 8 | 60 | 279 | 0.775000 | 44 | 16 | 194 | 85 | 0.773012 |
| 9 | 48 | 327 | 0.908333 | 31 | 17 | 225 | 102 | 0.906594 |
| 10 | 24 | 351 | 0.975000 | 11 | 13 | 236 | 115 | 0.974503 |
| 11 | 8 | 359 | 0.997222 | 4 | 4 | 240 | 119 | 0.997057 |
| 12 | 1 | 360 | 1.000000 | 0 | 1 | 240 | 120 | 1.000000 |

Table 3.2: Heuristic distributions for Manhattan distance on the Five Puzzle.

Table 3.2 shows the overall distribution for the Manhattan distance heuristic on the Five Puzzle. Manhattan distance is computed by counting the number of grid units that each tile is displaced from its goal position, and summing these values for all tiles. The first column of Table reftab:distribution gives the heuristic value. The second column gives the number of states of the Five Puzzle with each heuristic value. The third column gives the total number of states with a given or smaller heuristic value, which is simply the cumulative sum of the values from the second column. The fourth column gives the overall heuristic distribution $D(h)$. These values are computed by dividing the value in the third column by 360, the total number of states in the problem space. The remaining columns will be explained below.

The overall distribution is easily obtained for any heuristic. For heuristics implemented by table-lookup, or *pattern databases* [75, 215, 218], the distribution can be determined exactly by scanning the table. Alternatively, for a heuristic computed by a function, such as Manhattan distance on large sliding-tile puzzles, we can randomly sample the problem space to estimate the overall distribution to any desired degree of accuracy.

For heuristics that are the maximum of several different heuristics, we can approximate the distribution of the combined heuristic from the distributions of the individual heuristics by assuming that the individual heuristic values are independent.

The distribution of a heuristic function is not a measure of its accuracy, and says little about the correlation of heuristic values with actual costs. The only connection between the accuracy of a heuristic and its distribution is that given two admissible heuristics, the one with higher values will be more accurate than the one with lower values on average.

### The equilibrium distribution

While the overall distribution is the easiest to understand, the complexity of IDA* depends on a potentially different distribution. The *equilibrium distribution* $P(h)$ is defined as the probability that a node chosen randomly and uniformly among all nodes at a given depth of the brute-force search tree has heuristic value less than or equal to $h$, in the limit of large depth.

If all states of the problem occur with equal frequency at large depths in the search tree, then the equilibrium distribution is the same as the overall distribution. For example, this is the case with the Rubik's Cube search tree described in Section 3.2.2. In general, however, the equilibrium distribution may not equal the overall distribution. In the Five Puzzle, for example, the overall distribution assumes that all states, and hence all blank positions, are equally likely. As we saw in Section 3.2.3, however, at deep levels in the tree, the blank is in a side position in more than 1/3 of the nodes, and in a corner position in less than 2/3 of the nodes. In the limit of large depth, the equilibrium frequency of side positions is $f_s = f_{cs} + f_{ss} = 0.203113 + 0.150097 = 0.35321$. Similarly, the frequency of corner positions is $f_c = f_{cc} + f_{sc} = 0.274854 + 0.371936 = 0.64679 = 1 - f_s$ . Thus, to compute the equilibrium distribution, we have to take these equilibrium fractions into account. The fifth and sixth columns of Table 3.2, labelled "Corner" and "Side", give the number of states with the blank in a corner or side position, respectively, for each heuristic value. The seventh and eighth columns, labelled "Csum" and "Ssum", give the cumulative numbers of corner and side states with heuristic values less than or equal to each particular heuristic value. The last column gives the equilibrium distribution $P(h)$. The probability $P(h)$ that the heuristic value of a node is less than or equal to h is the probability that it is a corner node, 0.64679, times the probability that its heuristic value is less than or equal to $h$, given that it is a corner node, plus the probability that it is a side node, 0.35321, times the probability that its heuristic value is less than or equal to $h$, given that it is a side node. For example, $P(2) = 0.64679 \cdot (3/240) + 0.35321 \cdot (3/120) = 0.016915$. This differs from the overall distribution $D(2) = 0.016667$.

The equilibrium heuristic distribution is not a property of a problem, but of a problem space. For example, including the parent of a node as one of its children can affect the equilibrium distribution, by changing the equilibrium fractions of different types of states. When the equilibrium distribution differs from the overall distribution, it can still be computed from a pattern database, or by random sampling of the problem space, combined with the equilibrium fractions of different types of states, as illustrated above.

To provide some intuition behind our main result, Fig. 3.4 shows a schematic representation of a search tree generated by an iteration of IDA* on an abstract problem instance, where all edges have unit cost. The vertical axis represents the depth of a node, which is also its g value, and the horizontal axis represents the heuristic value of a node. Each box

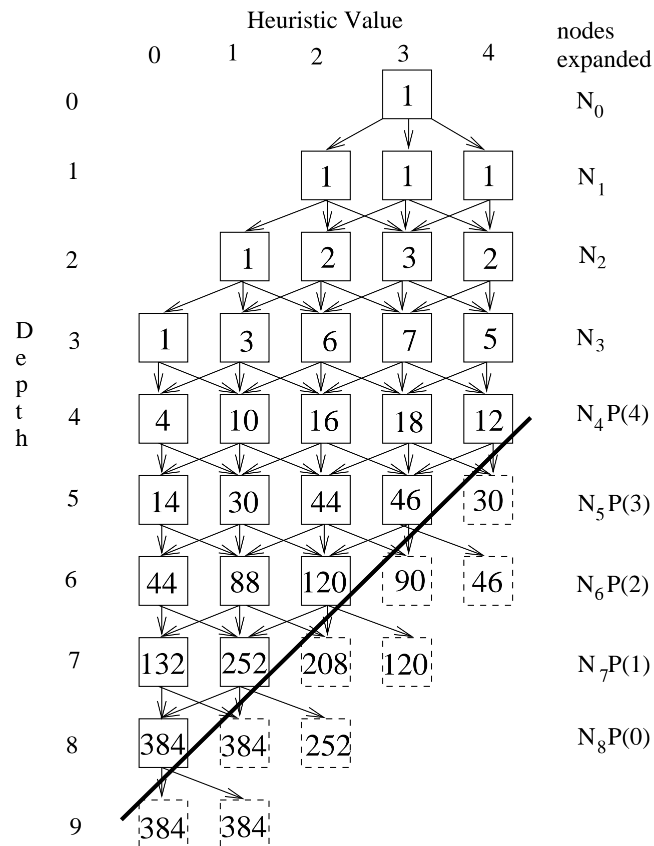


Figure 3.4: Sample tree for analysis of IDA*.

represents a set of nodes at the same depth with the same heuristic value, labelled with the number of such nodes. The arrows represent the relationship between parent and child node sets. These particular numbers were generated by assuming that each node generates one child each with heuristic value one less, equal to, and one greater than the heuristic value of the parent. For example, there are 6 nodes at depth 3 with heuristic value 1, 1 whose parent has heuristic value 1, 2 whose parents have heuristic value 2, and 3 whose parents have heuristic value 3. In this example, the maximum value of the heuristic is 4, and the heuristic value of the initial state is 3.

One assumption of our analysis is that the heuristic is consistent. Because of this, and since all edges have unit cost in this example, the heuristic value of a child must be at least the heuristic value of its parent, minus one. We assume a cutoff threshold of eight moves for this iteration of IDA* . Solid boxes represent sets of "fertile" nodes that will be expanded, while dotted boxes represent sets of "sterile" nodes that will not be expanded, because their total cost, $f(n) = g(n) + h(n)$ exceeds the cutoff threshold of 8. The thick diagonal line separates the fertile node sets from the sterile node sets.

### Nodes expanded as a function of depth

The values at the far right of Fig. 3.4 show the number of nodes expanded at each depth, which is the number of fertile nodes at that depth. $N_i$ is the number of nodes in the brute-force search tree at depth $i$, and $P(h)$ is the equilibrium heuristic distribution. The number of nodes generated is the branching factor times the number expanded.

Consider the graph from top to bottom. There is a root node at depth 0, which generates $N_1$ children. These nodes collectively generate $N_2$ child nodes at depth 2. Since the cutoff threshold is 8 moves, in the worst-case, all nodes n whose total cost $f(n) = g(n) + h(n) \leq 8$ will be expanded. Since 4 is the maximum heuristic value, all nodes down to depth $8 - 4 = 4$ will be expanded, Thus, for $d \leq 4$, the number of nodes expanded at depth $d$ will be $N_d$, the same as in a brute-force search. Since 4 is the maximum heuristic value, $P(4) = 1$, and hence $N_4P(4) = N_4$.

The nodes expanded at depth 5 are the fertile nodes, or those for which $f(n) = g(n) + h(n) = 5 + h(n) \leq 8$, or $h(n) \leq 3$. At sufficiently large depths, the distribution of heuristic values converges to the equilibrium distribution. Assuming that the heuristic distribution at depth 5 approximates the equilibrium distribution, the fraction of nodes at depth 5 with $h(n) \leq 3$ is approximately $P(3)$. Since all nodes at depth 4 are expanded, the total number of nodes at depth 5 is $N_5$, and the number of fertile nodes is $N_5P(3)$.

There exist nodes at depth 6 with heuristic values from 0 to 4, but their distribution does not equal the equilibrium distribution. In particular, nodes with heuristic values 3 and 4, shown in dotted boxes, are underrepresented relative to the equilibrium distribution, because these nodes are generated by parents with heuristic values from 2 to 4. At depth 5, however, the nodes with heuristic value 4 are sterile, producing no offspring at depth 6, hence reducing the number of nodes at depth 6 with heuristic values 3 and 4.

The number of nodes at depth 6 with $h(n) \leq 2$ is completely unaffected by any pruning however, since their parents are nodes at depth 5 with $h(n) \leq 3$, all of which are fertile. In other words, the number of nodes at depth 6 with $h(n) \leq 2$, which are the fertile nodes, is exactly the same as in the brute-force search tree, or $N_6P(2)$.

Due to consistency of the heuristic function, all possible parents of fertile nodes are themselves fertile. Thus, the number of nodes to the left of the diagonal line in Fig. 4 is exactly the same as in the brute-force search tree. In other words, heuristic pruning of the tree has no effect on the number of fertile nodes, although it does effect the sterile nodes. If the heuristic was inconsistent, then the distribution of fertile nodes would change at every level where pruning occurred, making the analysis far more complex.

When all edges have unit cost, the number of fertile nodes at depth $i$ is $N_iP(d-i)$, where $N_i$ is the number of nodes in the brute-force search tree at depth $i$, $d$ is the cutoff depth, and $P$ is the equilibrium heuristic distribution. The total number of nodes expanded by an iteration of IDA* to depth is

$$\sum_{i=0}^{d} N_iP(d-i).$$

### 3.3.6 General result

Here we state and prove our main theoretical result. First, we assume a minimum edge cost, and divide all costs by this value, normalizing it to one. We express all costs as multiples of the minimum edge cost. We allow operators with different costs, and replace the depth of a node by $g(n)$, the sum of the edge costs from the root to the node. Let $N_i$ be the number of nodes n in the brute-force search tree with $g(n) = i$.

Next, we assume the heuristic returns an integer multiple of the minimum edge cost. Given an admissible non-integer valued heuristic, we round up to the next larger integer, preserving admissibility. We also assume that the heuristic is consistent, meaning that for

any two nodes $n$ and $m$, $h(n) \leq k(n,m) + h(m)$, where $k(n,m)$ is the cost of an optimal path from $n$ to $m$.

Given these assumptions, our task is to determine $E(N,c,P)$, the number of nodes $n$ for which $f(n) = g(n) + h(n) \leq c$, given a problem-space tree with $N_i$ nodes of cost $i$, with a heuristic characterized by the equilibrium distribution $P(x)$. This is the number of nodes expanded by an iteration of IDA* with cost threshold $c$, in the worst case.

**Theorem 2** *In the limit of large* $c$ *,*

$$E(N,c,P) = \sum_{i=0}^{c} N_i P(c-i).$$

**Proof:** $E(N,c,P)$ is the number of nodes $n$ for which $f(n) = g(n) + h(n) \leq c$. Consider the nodes $n$ for which $g(n) = i$, which is the set of nodes of cost $i$ in *the brute-force search tree*. There are $N_i$ such nodes. The nodes of cost $i$ that will be expanded by IDA* in an iteration with cost threshold $c$ are those for which $f(n) = g(n) + h(n) = i + h(n) \leq c$, or $h(n) \leq c - i$. By definition of $P$, in the limit of large $i$, the number of such nodes in the brute-force search tree is $N_i P(c-i)$. It remains to show that all these nodes in the brute-force search tree are also in the tree generated by IDA*.

Consider an ancestor node $m$ of such a node $n$. Since $m$ is an ancestor of $n$, there is only one path between them in the tree, and $g(n) = i = g(m) + K(m,n)$, where $K(m,n)$ is the cost of the path from node $m$ to node $n$. Since $f(m) = g(m) + h(m)$, and $g(m) = i - K(m,n)$, $f(m) = i - K(m,n) + h(m)$. Since the heuristic is consistent, $h(m) \leq k(m,n) + h(n)$, where $k(m,n)$ is the cost of an optimal path from $m$ to $n$ in the problem graph. Since $K(m,n) \geq k(m,n)$, $h(m) \leq K(m,n) + h(n)$. Thus, $f(m) \leq i - K(m,n) + K(m,n) + h(n)$, or $f(m) \leq i + h(n)$. Since $h(n) \leq c - i$, $f(m) \leq i + c - i$, or $f(m) \leq c$. This implies that node $m$ is fertile and will be expanded during the search. Since all ancestors of node $n$ are fertile and will be expanded, node $n$ must eventually be generated. Therefore, all nodes $n$ in the brute-force search tree for which $f(n) = g(n) + h(n) \leq c$ are also in the tree generated by IDA*. Since there can't be any nodes in the IDA* tree that are not in the brute-force search tree, the number of such nodes at level $i$ in the IDA* tree is $N_i \leq P(c-i)$. Therefore, the total number of nodes expanded by IDA* in an iteration with cost threshold $c$ , which is the number in the last iteration, is

$$E(N,c,P) = \sum_{i=0}^{c} N_i P(c-i).$$

■

### 3.3.7 The heuristic branching factor

The effect of earlier iterations on the time complexity of IDA* depends on the rate of growth of node expansions in successive iterations. The *heuristic branching factor* is the ratio of the number of nodes expanded in a search to cost threshold $c$, divided by the nodes expanded in a search to cost $c-1$, or $E(N,c,P)/E(N,c-1,P)$, where 1 is the normalized minimum edge cost. Assume that the size of the brute-force search tree grows

exponentially with cost, or $N_i = b^i$, where is the brute-force branching factor. In that case, the heuristic branching factor $E(N, c, P)/E(N, c-1, P)$ is

$$\frac{\sum_{i=0}^{c} b^i P(c-i)}{\sum_{i=0}^{c-1} b^i P(c-1-i)} = \frac{b^0 P(c) + b^1 P(c-1) + b^2 P(c-2) + \cdots + b^c P(0)}{b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0)}.$$

The first term of the numerator, $b^0 P(c)$, is less than or equal to one, and can be dropped without significantly affecting the ratio. Factoring b out of the remaining numerator gives

$$\frac{b(b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0))}{b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0)} = b.$$

Thus, if the brute-force tree grows exponentially with branching factor $b$, then the running time of successive iterations of IDA* also grows by a factor of $b$. In other words, the heuristic branching factor is the same as the brute-force branching factor. In that case, it is easy to show that the overall time complexity of IDA* is $b/(b-1)$ times the complexity of the last iteration [213]. Previous analyses, based on different assumptions, predicted that the effect of a heuristic function is to reduce search complexity from $O(b^c)$ to $O(a^c)$, where $a < b$, reducing the effective branching factor [287]. Our analysis, however, shows that on an exponential tree, the effect of a heuristic is to reduce search complexity from $O(b^c)$ to $O(b^{c-k})$, for some constant $k$, which depends only on the heuristic function. If we define the effective depth of a search as the log base $b$ of the number of nodes expanded, where $b$ is the brute-force branching factor, then a heuristic reduces the effective depth from $c$ to $c-k$ for a constant $k$. In other words, a heuristic search to cost $c$ generates the same number of nodes as a brute-force search to cost $c-k$.

### 3.3.8 Experimental results

We tested our analysis experimentally by predicting the performance of IDA* on Rubik's Cube and sliding-tile puzzles, using well-known heuristics. Since all operators have unit cost in these problems, the $g(n)$ cost of a node $n$ is its depth. For $N_i$, we used the exact numbers of nodes at depth $i$, which were computed from the recurrence relations described in Section 3.2.3.

#### Rubik's Cube

We first predicted existing data on Rubik's Cube [215]. The problem space, described in Section 3.2.2, allows 180-degree twists as single moves, disallows two consecutive twists of the same face, and only allows consecutive twists of opposite faces in one order. This search tree has a brute-force branching factor of about 13.34847. The median optimal solution depth is 18 moves.

The heuristic is the maximum of three different pattern databases [75, 215]. It is admissible and consistent, with a maximum value of 11 moves, and an average value of 8.898 moves. The distribution of the individual heuristics was calculated exactly by scanning the databases, and the three heuristics were assumed to be independent to calculate the distribution of the combined heuristic. In this case, the equilibrium distribution is the same as the overall distribution. We ignored goal states, completing each search iteration.

| Depth | Theoretical | Problems | Experimental | Error |
|---|---|---|---|---|
| 10 | 1,510 | 1000 | 1,501 | 0.596% |
| 11 | 20,169 | 1000 | 20,151 | 0.089% |
| 12 | 269,229 | 1000 | 270,396 | 0.433% |
| 13 | 3,593,800 | 100 | 3,564,495 | 0.815% |
| 14 | 47,971,732 | 100 | 47,916,699 | 0.115% |
| 15 | 640,349,193 | 100 | 642,403,155 | 0.321% |
| 16 | 8,547,681,506 | 100 | 8,599,849,255 | 0.610% |
| 17 | 114,098,463,567 | 25 | 114,773,120,996 | 0.591% |

Table 3.3: Nodes generated by IDA* on Rubik's Cube

In Table 3.3, the first column shows the cutoff depth, the next column gives the node generations predicted by our theory, the next column indicates the number of problem instances run, the next column displays the average number of nodes generated by IDA* in a single iteration, and the last column shows the error between the theoretical prediction and experimental results.

The theory predicts the data to within 1% accuracy in every case. Sources of error include the limited number of problem instances, the assumption of independence of the heuristics, and the fact that the heuristic distribution at a finite depth does not equal the equilibrium distribution. The ratio between the node generations in the last two levels, which is the experimental heuristic branching factor, is 13.34595, compared to the theoretical value of 13.34847. If we take the log, base 13.34847, of the predicted number of nodes generated at depth 17 (114,098,463,567), we get about 9.825. Thus, this particular heuristic reduces the effective depth of search by $17 - 9.825 = 7.175$ moves.

**Eight Puzzle**

We also experimented with the Eight Puzzle, using the Manhattan distance heuristic. It has a maximum value of 22 moves, and a mean value of 14 moves. The optimal solution length averages 22 moves, with a maximum of 31 moves, assuming the blank is in a corner in the goal state. Since the Eight Puzzle has only 181,440 solvable states, the heuristic distributions were computed exactly. Three distributions were used, depending on whether the blank is in a center, corner, or side position. The number of nodes of each type at each depth of the brute-force tree was also computed exactly.

Table 3.4 shows a comparison of the number of node expansions predicted by our theoretical analysis, to the number of nodes expanded by a single iteration of IDA* to various depths, ignoring goal states. Each data point is the average of all 181,440 problem instances. Since the average numbers of node expansions, the size of the brute-force tree, and the heuristic distributions are all exact, the model predicts the experimental data exactly, to multiple decimal places, verifying that we have accounted for all the relevant factors.

The Eight Puzzle has even and odd-depth brute-force branching factors of 1.5 and 2. The corresponding heuristic branching factors are 1.667 and 1.8, but the product of the two branching factors is 3 in both cases. If we take the log, base $\sqrt{3}$, of the number of nodes expanded at depth 31 (160,167), we get about 21.8. This implies that on the Eight

| Depth | Theoretical | Problems | Experimental | Error |
|---|---|---|---|---|
| 20 | 393 | 181,440 | 393 | 0.0% |
| 21 | 657 | 181,440 | 657 | 0.0% |
| 22 | 1,185 | 181,440 | 1,185 | 0.0% |
| 23 | 1,977 | 181,440 | 1,977 | 0.0% |
| 24 | 3,561 | 181,440 | 3,561 | 0.0% |
| 25 | 5,936 | 181,440 | 5,936 | 0.0% |
| 26 | 10,686 | 181,440 | 10,686 | 0.0% |
| 27 | 17,815 | 181,440 | 17,815 | 0.0% |
| 28 | 32,072 | 181,440 | 32,072 | 0.0% |
| 29 | 53,450 | 181,440 | 53,450 | 0.0% |
| 30 | 96,207 | 181,440 | 96,207 | 0.0% |
| 31 | 160,167 | 181,440 | 160,167 | 0.0% |

Table 3.4: Nodes generated by IDA* on the Eight Puzzle

Puzzle, Manhattan distance reduces the effective depth of search by $31 - 21.8 = 9.2$ moves.

### Fifteen Puzzle

We ran a similar experiment on the Fifteen Puzzle, also using the Manhattan distance heuristic. The average heuristic value is about 37 moves, and the maximum is 62 moves. The average optimal solution length is 52.5 moves. Since the Fifteen Puzzle has over $10^{13}$ solvable states, we used a random sample of ten billion solvable states to approximate the heuristic distributions. Three different distributions were used, one for the blank in a middle, corner, or side position. The number of nodes of each type at each depth was also computed exactly, for each different initial blank position.

| Depth | Theoretical | Problems | Experimental | Error |
|---|---|---|---|---|
| 40 | 42,664 | 100,000 | 41,973 | 1.65% |
| 41 | 90,894 | 100,000 | 91,495 | 0.66% |
| 42 | 193,641 | 100,000 | 191,219 | 1.27% |
| 43 | 412,535 | 100,000 | 415,490 | 0.72% |
| 44 | 878,864 | 100,000 | 870,440 | 0.96% |
| 45 | 1,872,330 | 100,000 | 1,886,363 | 0.75% |
| 46 | 3,988,805 | 100,000 | 3,959,729 | 0.73% |
| 47 | 8,497,734 | 100,000 | 8,562,824 | 0.77% |
| 48 | 18,103,536 | 100,000 | 18,003,959 | 0.55% |
| 49 | 38,567,693 | 100,000 | 38,864,269 | 0.77% |
| 50 | 82,164,440 | 100,000 | 81,826,008 | 0.41% |

Table 3.5: Nodes generated by IDA* on the Fifteen Puzzle

Table 3.5 is similar to Table 3.4. Each line is the average of 100,000 random solvable problem instances. Despite over ten orders of magnitude variation in the nodes expanded

in individual problem instances, the average values agree with the theoretical prediction to within 1% in most cases. The ratio between the experimental number of node expansions at the last two depths is $2.105$, compared to the brute-force branching factor of $2.130$. If we take the log, base $2.13$, of the predicted number of nodes expanded at depth 50 (82,164,440), we get about $24.1$. Thus, on the Fifteen Puzzle, Manhattan distance reduces the effective depth of search by $50 - 24.1 = 25.9$ moves.

The results above are for single complete iterations to the given search depths, ignoring any solutions found. How well do these results predict the running time of IDA* to solve a random problem instance? The average optimal solution length for random Fifteen Puzzle instances is about 52.5 moves [218]. Multiplying the last value in Table 3.5 by $b^2$ or $2.13042^2$ predicts 372,911,869 node expansions in a complete iteration to depth 52, or 794,451,446 node generations. Multiplying by $b^2/(b^2 - 1) = 1.2826$ to account for all the previous iterations predicts about 1.019 billion node generations. Completing the final iteration to find all optimal solutions to the same set of 1000 problem instances generates an average of 1.178 billion nodes. Terminating IDA* when the first solution is found generates an average of 401 million nodes.

### Twenty-Four Puzzle

We can also predict the performance of IDA* on problems we can't run experimentally, such as the Twenty-Four Puzzle with the Manhattan distance heuristic. The brute-force branching factor is 2.36761. Sampling ten billion random solvable states yields an approximation of the overall heuristic distribution, which approximates the equilibrium distribution. The average heuristic value is 76 moves. Experiments using more powerful disjoint pattern database heuristics [218] give an average optimal solution length of about 100 moves. Our theory predicts that running all iterations up to depth 100 will generate an average of $1.217 \times 10^{19}$ nodes. On a 440 MHz Sun Ultra 10 workstation, IDA* with Manhattan distance generates about 7.5 million nodes per second. This predicts an average time to complete all iterations up to depth 100, on a random instance of the Twenty-Four Puzzle, ignoring any solutions found, of about 50,000 years! Manhattan distance reduces the effective depth of search on the Twenty-Four Puzzle by about 49 moves.

### Observed heuristic branching factor

If we run IDA* on a single instance of a sliding-tile puzzle, we observe that the ratio between the numbers of nodes generated in successive iterations usually decreases with each iteration, but exceeds the theoretical heuristic branching factor. On the sliding-tile puzzles with Manhattan distance, the cost threshold increases by two in each successive iteration, and hence the theoretical heuristic branching factor is the square of the brute-force branching factor. For example, in the Twenty-Four Puzzle, the observed heuristic branching factor is often greater than 10, whereas $b^2$ is only 5.6.

The reason for this discrepancy is an initial transient in the observed heuristic branching factor. The formula in Theorem 2 is based on the equilibriumheuristic distribution. Starting from a single initial state, it takes many iterations of IDA* for the heuristic distribution to converge to the equilibrium distribution. This effect is ameliorated in the results presented above because the experimental data is averaged over a large number of initial states. If we run IDA* long enough on a single problem instance, the observed heuristic

branching factor eventually converges to the square of the brute-force branching factor.

Why is the observed heuristic branching factor greater than the theoretical branching factor? The heuristic distribution at the root of the search tree starts with the heuristic value of the initial state, and gradually spreads out to larger and smaller heuristic values with increasing depth. Thus, the frequency of small and large heuristic values is initially zero, underestimating their frequency in the equilibrium heuristic distribution. Underestimating the large values has little effect, since the frequency of these values is multiplied by the relatively small number of nodes at shallow depths. The frequencies of small values, however, are multiplied by the large numbers of nodes at deep depths, and hence underestimating these values significantly decreases the number of node generations, relative to what happens at equilibrium. As the search depth increases in successive iterations, the frequency of nodes with small heuristic values increases, which causes a larger observed heuristic branching factor than occurs at equilibrium.

In Rubik's Cube, however, the observed heuristic branching factor converges to the brute-force value, without consistently overestimating it initially. This is due to the smaller range of heuristic values, and the larger branching factor, which allows convergence to the equilibrium heuristic distribution more quickly.

## 3.4 Conclusions

We first show how to compute the exact number of nodes at different depths, and the asymptotic branching factor, of brute-force search trees where different nodes have different numbers of children. We begin by writing a set of recurrence relations for the generation of the different node types. By expanding these recurrence relations, we can determine the exact number of nodes at a given depth, in time linear in the depth. We can also use the ratio of the numbers of nodes at successive depths to approximate the asymptotic branching factor with very high precision. Alternatively, we can rewrite the recurrence relations as a set of simultaneous equations involving the relative frequencies of the different types of nodes, and solve them analytically for small numbers of node types. We give the asymptotic branching factors for Rubik's Cube, the Five Puzzle, and the first nine square sliding-tile puzzles.

We then use these results to predict the time complexity of IDA*. We characterize a heuristic by the distribution of heuristic values, which can be obtained by random sampling, for example. We compare our predictions with experimental data on Rubik's Cube, the Eight Puzzle, and the Fifteen Puzzle, getting agreement within 1% for Rubik's Cube and the Fifteen Puzzle, and exact agreement for the Eight Puzzle. In contrast to previous results, our analysis and experiments indicate that on an exponential tree, the asymptotic heuristic branching factor is the same as the brute-force branching factor. Thus, the effect of a heuristic is to reduce the effective depth of search by a constant, relative to a brute-force search, rather than reducing the effective branching factor.

## 3.5 Generality and further work

To what extent can these results can be applied to other problems? Our main result is Theorem 2. It says that the number of nodes n for which $f(n) = g(n) + h(n) \leq c$ is a convolution of two distributions. The first is the number of nodes of a given cost in the

brute-force search space, and the second is the number of nodes with a given heuristic value. In order to apply this to a particular problem, however, we have to determine the size of the brute-force search space, and the heuristic distribution. Thus, we have decomposed the problem of predicting the performance of a heuristic search algorithm into two simpler problems.

How could we use this analysis to predict the performance of A*? The main difference between A* and IDA* is that A* detects duplicate nodes, and doesn't reexpand them. Theorem 2 applies to A* as well, but $N_i$ is the number of nodes in the problem-space graph, rather than its tree expansion. Unfortunately, the only technique known for computing the number of nodes at a given depth in a search graph is exhaustive search to that depth. As a result, these values are unknown for even regular problem spaces such as the Fifteen Puzzle or Rubik's Cube. The relevant heuristic distribution $P(h)$ for analyzing A* is the overall heuristic distribution $D(h)$, because each state occurs only once in the problem space.

As another example, could we predict the performance of IDA* on the traveling salesman problem? In a problem space that constructs a tour by adding one city at a time, each node represents a partial tour, and the number of nodes at depth is the number of permutations of $n-1$ elements taken at a time. Computing the distribution for a heuristic such as the cost of a minimum spanning tree of the remaining cities is difficult, however. It depends on the depth of search, and the particular problem instance. If the edge costs and heuristic values are real numbers rather than integers, the discrete convolution of Theorem 2 becomes a continuous convolution, and the summation becomes an integral. While we can't solve this problem currently, Theorem 2 tells us what distributions we need, and how to combine them.

The running time of IDA* depends on the branching factor, the heuristic distribution, and the optimal solution cost. Predicting the optimal solution cost for a given problem instance, or even the average optimal solution cost, is an open problem, however. Since the number of nodes in a problem-space tree grows by a factor of $b$ with each succeeding depth, a lower bound on the maximum optimal solution depth is the log base $b$ of the number of reachable states, rounded up to the next larger integer. This can be used as an estimate of the average solution depth. For example, this method predicts a depth of 22 moves for the Eight Puzzle, which equals the average optimal solution length. For Rubik's Cube, this method predicts a value of 18 moves, which is the median optimal solution length. For the Fifteen Puzzle, however, we get an estimate of only 40 moves, while the average solution depth is 52.5 moves. The reason this method doesn't accurately predict the maximum solution depth is that it assumes that all states in the search tree are unique. For all these problems, however, there are multiple paths to the same state, giving rise to duplicate nodes in the tree representing the same state.

# Paper 4

# Prediction of Regular Search Tree Growth by Spectral Analysis

S. Edelkamp.
Institut für Informatik
Georges-Köhler-Allee, Gebäude 51
79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science. Springer, pages 154–168, 2001.

**Abstract**

The time complexity analysis of the IDA* algorithm has shown that predicting the growth of the search tree essentially relies on only two criteria: The number of nodes in the brute-force search tree for a given depth and the equilibrium distribution of the heuristic estimate. Since the latter can be approximated by random sampling, we accurately predict the number of nodes in the brute-force search tree for large depth in closed form by analyzing the spectrum of the problem graph or one of its factorization.

We further derive that the asymptotic brute-force branching factor is in fact the spectral radius of the problem graph and exemplify our considerations in the domain of the $(n^2 - 1)$-Puzzle.

## 4.1 Introduction

Heuristic search is essential to AI, since it allows very large problem spaces to be traversed with a considerably small number of node expansions. Nevertheless, storing this number of nodes in memory, as required in the A* algorithm [161], often exceeds the resources available. This is bypassed in an iterative deepening version of A*, IDA* for short, that searches the tree expansion of the original state graph instead of the graph itself. IDA* [213] applies bounded depth-first traversals with an increasing threshold on A*'s node evaluation function. The tree expansion may contain several duplicate nodes such that low memory consumption is counterbalanced with a considerably high overhead in time.

Fortunately, due to simple search tree pruning rules and expressive heuristic estimates to direct the search process, duplicates in regular search spaces are rare such that IDA* has been very successfully applied to solve solitaire games like the $(n^2 - 1)$ Puzzle [213, 218, 221] and Rubik's Cube [215].

Korf, Reid and Edelkamp [220] have analyzed the IDA* algorithm to predict the search performance of IDA* in the number of node expansions for a specific problem. The main result is that assuming consistency[1] of the integral heuristic estimate in the limit of large $c$, the expected total number of node expansions with cost threshold $c$ in one iteration of IDA* is equal to

$$\sum_{d=0}^{c} n^{(d)} P(c-d),$$

where $n^{(d)}$ is the number of nodes in the brute-force search tree with depth $d$ and $P$ is the equilibrium distribution defined as the probability distribution of heuristic values in the limit of large depth. More precisely, $P(h)$ is the probability that a randomly and uniformly chosen node of a given depth has a heuristic value less than or equal to $h$. In practice the equilibrium distribution for admissible heuristic functions will be approximated by random sampling [219]; a representative sample of the problem space is drawn and classified according to the integral heuristic evaluation function. The value $n^{(d)}$ for large depths $d$ without necessarily exploring the search tree, can be approximated with the asymptotic brute-force branching factor; the number of nodes at one depth divided by the number of nodes in the next shallower depth, in the limit as the depth goes to infinity. The asymptotic heuristic branching factor is defined analogously on search tree levels for two occurring values on the node evaluation function $f$. In some domains we observe anomalies in the limiting behavior of the asymptotic branching factors, e.g., in the $(n^2 - 1)$-Puzzle and odd values of $n$ it alternates between two different values [104].

The observation that a consistent heuristic estimate $h$ affects the relative depth to a goal instead of the branching itself is supported by the fact that IDA*'s exploration is equivalent to undirected iterative deepening exploration in a re-weighted problem graph with costs $1 + h(v) - h(u)$ for all edges $(u, v)$. The new node evaluation $f'(u_j)$ of node $u_j$ on path $p = (s = u_1, \ldots, u_t = t)$ equals $\sum_{i=1}^{j-1}(1 + h(u_{i+1}) - h(u_i))$ and telescopes to the old merit $f(u_j)$ minus $h(s)$. Therefore, the heuristic is best understood as a bonus to

[1]Consistent heuristic estimates satisfy $h(v) - h(u) + 1 \geq 0$ for each edge $(u, v)$ in the underlying problem graph. They yield monotone node evaluations $f(u) = g(u) + h(u)$ on generating paths with length $g(u)$. Admissible heuristics are lower bound estimates that underestimate the goal distance for each state. Consistent estimates are admissible.

the search depth. Moreover, since we have only altered edge weights, it is not surprising that for bounded heuristic estimates and large depth the asymptotic heuristic branching factor equals the asymptotic brute-force branching factor.

Our main result in this paper is that in undirected problem graphs the value of the number of nodes in depth $d$ of the brute-force search can be computed effectively by analyzing the spectrum of the adjacency matrix for the problem graph. The analysis requires some results of linear algebra and an algorithm of applied mathematics. Since the problem graph is considered to be large for regular search spaces we show how to factorize the problem graph through an equivalence relation of same branching behavior. We take the $(n^2 - 1)$-Puzzle as the running example, discuss the generality of the results from various points of view: other problem domains, general, especially undirected graph structures, and predecessor pruning. Finally, we give concluding remarks and shed light on future research options.

## 4.2 Linear Algebra Basics

**Linear Mappings and Bases** A mapping $f : V \to W$, with $V, W$ being vector spaces over the field $K$ (e.g. the set of real or the set of complex numbers) is *linear*, if $f(\lambda v + \mu w) = \lambda f(v) + \mu f(w)$ for all $v, w \in V$ and all $\lambda, \mu \in K$. A *basis* of a vector space $V$ is a linear independent set of vectors that spans $V$. If the basis is finite, its cardinality defines the dimension $dim(V)$ of the vector space $V$, otherwise the dimension is said to be infinite.

**Matrices and Basis-Transformations** Linear mappings of vector spaces of finite dimension can be represented as matrices, since there is an isomorphism that maps the set of all $(m \times n)$ matrices to the set of all linear mappings from $V$ to $W$ according to their respectively fixed bases, where $dim(V) = n$ and $dim(W) = m$. Usually, $V$ equals $W$ and in this case the linear mapping $f$ is called *endomorphism.* A *basis-transformation* from basis $\mathcal{A}$ to $\mathcal{B}$ in the vector space $V$ can be represented by a transformation matrix $C_{\mathcal{AB}}$ which is the inverse of $C_{\mathcal{BA}}$. Very often, $\mathcal{A}$ is the canonical basis. Computing the inverse $C^{-1}$ of a matrix $C$ can be achieved by elementary row transformations, that convert the $(n \times 2n)$ matrix $[C \mid I]$ into $[I \mid C^{-1}]$, with $I$ being the identity matrix.

**Similarity and Normal Forms** Two matrices $A$ and $B$ are *similar*, if there is a matrix $C$ with $B = CAC^{-1}$. This is equivalent to the fact that there is an endomorphism $f$ of $V$ and two bases $\mathcal{A}$ and $\mathcal{B}$ with matrix $A$ representing $f$ according to $\mathcal{A}$ and $B$ representing $f$ according to $\mathcal{B}$. Similarity is an equivalence relation and one main problem in linear algebra is to derive a concise representative in the equivalence class of similar matrices, the *normal form.* A very simple form is the diagonal shape with non-zero values $\lambda_1, \ldots, \lambda_n$ only on the main diagonal. In this case, a matrix $B$ is called *diagonizable* and can be written as $B = C \cdot \text{diag}(\lambda_1, \ldots, \lambda_n) \cdot C^{-1}$. Unfortunately, not all matrices are diagonizable, especially when the linear mapping is defined on the set of real numbers. Even if the vector space defining field is the set of complex numbers, only *tridiagonizability* can be granted, in which matrix $A$ may have non-zero components above the main diagonal. Further simplifications lead to the so-called *Jordan normal form.*

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | |

Figure 4.1: The Eight-, Fifteen- and Twenty-Four-Puzzles.

**Eigenvalues and Eigenspaces** An endomorphism $f$ of a vector space $V$ over the field $K$ contains an *eigenvalue* $\lambda \in K$, if there is a non-trivial vector $v \in V$, with $f(v) = \lambda v$. Any such non-trivial vector $v \in V$ with $f(v) = \lambda v$ is called *eigenvector*. If there is a basis $\mathcal{B}$ of eigenvectors, then the matrix representation according to $\mathcal{B}$ has a diagonal shape. In this case $f$ is also called *diagonizable*. It can be shown that the eigenvalues are roots of the *characteristic equation* $P_f(\lambda) = \det(A - \lambda I) = 0$, where the determinant det(A) is defined as $\sum_{\sigma \in S_n} \prod_{i<j} (\sigma(j) - \sigma(i))/(j-i)) \cdot a_{1\sigma(1)} \cdot \ldots \cdot a_{n\sigma(n)}$ with $S_n$ being the set of all $n$-permutations. If the polynomial $P_f(\lambda)$ factorizes, i.e. $P_f(\lambda) = \mathit{const} \cdot \prod_{i=1}^{k} (\lambda - \lambda_i)$, which is the case for matrices of complex numbers, the corresponding eigenspaces $E_f(\lambda_i)$ have to be computed. If then the number of occurring linear terms $(\lambda - \lambda_i)$ in $P_f(\lambda)$, the *algebraic multiplicity* of $\lambda_i$, equals the dimension of $E_f(\lambda_i)$, the *geometric multiplicity* of $\lambda_i$, then $A$ is indeed *diagonizable*.

## 4.3 Partitioning the Search Space

The $(n^2 - 1)$-Puzzle is a sliding tile toy problem. It consists of $(n^2 - 1)$ numbered tiles that can be slid into a single empty position, called the blank. The goal is to rearrange the tiles such that a certain goal position is reached. Figure 4.1 depicts possible end configurations of well-known instances to the $(n^2 - 1)$-Puzzle: For $n = 3$ we get the Eight-Puzzle, for $n = 4$ the Fifteen-Puzzle and for $n = 5$, the Twenty-Four-Puzzle is met. The state spaces for these problems grow exponentially. The exact number of reachable states (independent of the initial one) is $(n^2)!/2$ which resolves to approximately $10^5$ states for the Eight-Puzzle, $10^{13}$ states in the Fifteen-Puzzle and $10^{25}$ states in the Twenty-Four-Puzzle.

We partition the search space $S$ in classes $S_1, \ldots, S_k$, collecting states into groups with same branching behavior. In other words we devise an equivalence relation that partitions the state space into equivalence classes: two states are equivalent if their long term branching behavior coincides. All states in one equivalence class $S_i$, $i \in \{1, \ldots, k\}$, necessarily have the same node branching factor, defined as the number of children a node has in the brute-force search tree.

For the example of the $(n^2 - 1)$-Puzzle a partition is given by the following relation: two states are equivalent if the blank is at the same absolute position. Obviously the subtrees of such nodes are isomorphic, since the branching behavior of equivalent states has to be the same. A further reduction of the search tree is established by partitioning the search space with respect to symmetry. For the $(n^2 - 1)$ Puzzle we establish three branching types: corner or $c$-nodes with node branching factor 2, side or $s$-nodes with

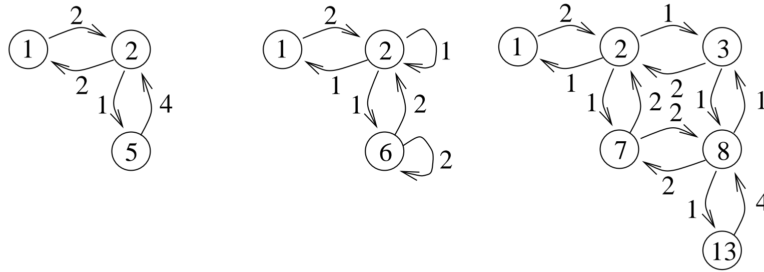


Figure 4.2: Equivalence Graph for the Eight-, Fifteen- and Twenty-Four-Puzzles.

node branching factor 3, and middle or $m$-nodes with node branching factor 4. However, does the long time node branching behavior depend on these node types only? In the Eight- and Fifteen-Puzzles this is the case, since for symmetry reasons all $c$, $s$ and $m$ nodes generate the same subtree structure. For the Twenty-Four-Puzzle, however, the search tree of two side or two middle states may differ. For this case we need six classes with a blank at position 1,2,3,7,8, and 13 according to the tile labeling in Figure 4.2. In the general case the number of different node branching classes in the $(n^2-1)$ Puzzle is

$$\sum_{i=0}^{\lceil n/2\rceil} i = \binom{\lceil n/2\rceil}{2} = \lceil n/2\rceil(\lceil n/2\rceil - 1)/2.$$

This still compares well to a partition according to the $n^2$ equivalent classes in the first factorization (savings of a factor of about eight) and of course to the $(n^2)!/2$ states in the overall search space (exponential savings).

## 4.4 Equivalence Graph Structure

Utilizing this partition technique we define the weighted *equivalence graph* $\overline{G} = (\overline{V}, \overline{E}, w)$ as follows. The set of nodes $\overline{V}$ equals the set of equivalence classes and an edge $e$ from class $S_i \in \overline{V}$ to $S_j \in \overline{V}$ with weight $w(e)$ is drawn, if every state in $S_i$ leads to $w$ states in class $S_j$. Obviously, the sum of all outgoing edges equals the node branching factor. Let $A_{\overline{G}}$ be the adjacency matrix with respect to the *equivalence graph* $\overline{G}$. Since the explorations in $G$ and $\overline{G}$ span the same search-tree structure the search tree growth will be the same.

A generator matrix $P$ for the population of nodes according to the given equivalence relation is defined by $P = A^T_{\overline{G}}$. More precisely, $P_{j,i} = l$ if a node of type $i$ in a given level leads to $l$ nodes of type $j$ in the next level. We immediately infer that $N^{(d)} = PN^{(d-1)}$, with $N^{(d)}$ being the vector of nodes in depth $d$ of the search tree. If $||\cdot||_1$ denotes the vector norm $||x||_1 = |x_1| + \ldots + |x_k|$ then the number of nodes $n^{(d)}$ in depth $d$ is equal to $||N^{(d)}||_1$.

The asymptotic branching factor $b$ (if it exists) is defined as the limit of $n^{(d)}/n^{(d-1)}$ for increasing $d$ and equals the weighted product of the node frequencies $b = \sum_{i=1}^k b_i f_i$, where $f_i$ is the fraction of nodes of class $i$ with respect to the total number of nodes. As we will see, we can compute the branching factor analytically without actually determining node frequency values.

The first observation is that in case of convergence the asymptotic branching factor is not only met in the the overall search tree expansion but in every equivalence class. Since all frequencies of nodes converge we have that $b = \lim_{d\to\infty} N_i^{(d)}/N_i^{(d-1)}$, with $N_i^{(d)}$ being the number of nodes of class $i$ in depth $d$, $i \in \{1,\ldots,k\}$. In other words, if the ratio of the cardinality of one equivalence class and the overall search space size settles and the search space size grows with factor $b$, then the equivalence class size itself grows with factor $b$.

We represent the fractions $f_i$ as a distribution vector $F$. We first assume that this vector converges in the limit of large depth. The considerations for an analytical solution to the branching factor problem result in the equations $bF = FP$, where $b$ is the asymptotic branching factor. In addition, we have the equation that the total of all node frequencies is one. The underlying mathematical issue is an eigenvalue problem. Transforming $bF = PF$ leads to $0 = (P - bI)F$ for the identity matrix $I$. The solutions for $b$ are the roots of the characteristic equation $\det(P - bI) = 0$ where $\det$ is the determinant of the matrix. Since $\det(P - bI) = \det(P^T - bI)$ the transposition of the equivalence graph matrix $A_{\overline{G}}$ preserves the value of $b$. In case of the Eight-Puzzle $\det(P - bI)$ equals

$$\det\begin{pmatrix} 0-b & 2 & 0 \\ 2 & 0-b & 1 \\ 0 & 4 & 0-b \end{pmatrix} = 0.$$

This equation is equivalent to $b(4-b^2)+4b = 0$, yielding the following three solutions $-\sqrt{8} = -2.828427124$, $0$, $\sqrt{8} = 2.828427124$. Experimental results show that the branching factor alternates every two depth values between 3 and $8/3 = 2.666666666$. Since $\sqrt{8}$ is the geometric mean of 3 and $8/3$ the value $\sqrt{8}$ is the proper choice for the asymptotic branching factor $b$ of the brute-force search tree.

For the case of the Fifteen-Puzzle we have to calculate

$$\det\begin{pmatrix} 0-b & 2 & 0 \\ 1 & 1-b & 1 \\ 0 & 2 & 2-b \end{pmatrix} = 0,$$

which simplifies to $(1-b)(b-2)b + 4b - 4 = 0$. The solution to this equation are $1$, $1+\sqrt{5} = 3.236067978$, and $1-\sqrt{5} = -1.236067978$. The value $1+\sqrt{5}$ matches experimental data for the asymptotic branching factor.

For the Twenty-Four-Puzzle we have to solve

$$\det\begin{pmatrix} 0-b & 2 & 0 & 0 & 0 & 0 \\ 1 & 0-b & 1 & 1 & 0 & 0 \\ 0 & 2 & 0-b & 0 & 1 & 0 \\ 0 & 2 & 0 & 0-b & 2 & 0 \\ 0 & 0 & 1 & 2 & 0-b & 1 \\ 0 & 0 & 0 & 0 & 4 & 0-b \end{pmatrix} = 0.$$

The six eigenvalues are $0$, $0$, $\sqrt{3} = 1.732050808$, $-\sqrt{3} = -1.732050808$, $\sqrt{12} = 3.464101616$, and $-\sqrt{12} = -3.464101616$. Experiments show that for large depth the branching factor oscillates and that the geometric mean is $3.464101616$.

We conclude that the asymptotic branching factor in the example problems is the *largest* eigenvalue of the adjacency matrix for the equivalence graph and that we observe

anomalies if the largest eigenvalue has a negative counterpart of the same absolute value. In the following we will analyze the structure of the eigenvalue problem to show why this is the case.

## 4.5 Exact Prediction of Search Tree Size

The equation $N^{(d)} = PN^{(d-1)}$ can be unrolled to $N^{(d)} = P^d N^{(0)}$. We briefly sketch how to compute $P^d$ for large $d$. We have seen that $P$ is *diagonizable*, if there exists a invertible matrix $C$ and a diagonal matrix $Q$ with $P = CQC^{-1}$. This simplifies the calculation of $P^d$, since we have $P^d = CQ^dC^{-1}$ (the remaining terms $C^{-1}C$ cancel). By the diagonal shape of $Q$ the value of $Q^d$ is obtained by taking the matrix elements $q_{i,i}$ to the power of $d$. These elements are the eigenvalues of $P$. This connection is not surprising, since in case of convergence of the vector of node frequencies $F$ we have seen that the branching factor itself is a solution to the eigenvalue problem $PF = bF$. We conclude that in case of *diagonizability* we can exactly predict the number of nodes of depth $d$ by determining the set of eigenvalues of $P$.

In the example of the Eight-Puzzle the eigenvectors for the eigenvalues $-\sqrt{8}$, $0$, and $\sqrt{8}$ are $(2, -\sqrt{8}, 1)^T$, $(-2, 0, 1)^T$, and $(2, \sqrt{8}, 1)^T$, respectively. Therefore, the basis-transformation matrix $C$ is given by

$$C = \begin{pmatrix} 2 & -2 & 2 \\ -\sqrt{8} & 0 & \sqrt{8} \\ 1 & 1 & 1 \end{pmatrix}$$

with the following inverse

$$C^{-1} = 1/16 \begin{pmatrix} 2 & -\sqrt{8} & 4 \\ -2 & 0 & 8 \\ 2 & \sqrt{8} & 4 \end{pmatrix}.$$

With $Q = diag(-\sqrt{8}, 0, \sqrt{8})$ we have $C^{-1}C = I$ and $C^{-1}PC = Q$. Therefore, calculating $N^{(d)} = P^d N^{(0)}$ for $d \geq 1$ corresponds to $N^{(d)} = CQ^dC^{-1}N^{(0)}$, where $Q^d = \text{diag}((-\sqrt{8})^d, 0, (\sqrt{8})^d)$. Hence, $N^{(d)}$ equals to

$$1/16 \begin{pmatrix} 2 & -2 & 2 \\ -\sqrt{8} & 0 & \sqrt{8} \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} (-\sqrt{8})^d & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & (\sqrt{8})^d \end{pmatrix} \begin{pmatrix} 2 & -\sqrt{8} & 4 \\ -2 & 0 & 8 \\ 2 & \sqrt{8} & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

which resolves to

$$N^{(d)} = 1/16 \left(4\sqrt{8}^d((-1)^d + 1), 2\sqrt{8}^{d+1}((-1)^{d+1} + 1), 2\sqrt{8}^d((-1)^d + 1)\right)^T.$$

The exact formula for $N^{(d)}$ and small values of $d$ validates the observed search tree growth: $N^{(1)} = (0, 2, 0)^T$, $N^{(2)} = (4, 0, 2)^T$, $N^{(3)} = (0, 16, 0)^T$, $N^{(4)} = (32, 0, 16)^T$, etc.

The closed form for $N^{(d)}$ explicitly states that the asymptotic branching factor for the Eight Puzzle is $\sqrt{8}$. Moreover, the odd-even effect for branching in that puzzle is established by the factor $(-1)^d + 1$, which cancels for an odd value of $d$. Nevertheless,

solving the characteristic equation and establishing the basis of eigenvectors by hand is tedious work. Fortunately, the application of symbolic mathematical tools such as Maple and Mathematica help to perform the calculations in larger systems.

For the Fifteen-Puzzle the basis-transformation matrix $C$ and its inverse $C^{-1}$ are

$$C = \begin{pmatrix} 1 & -1 & 1 \\ 1-\sqrt{5} & -1 & 1+\sqrt{5} \\ 3/2-1/2\sqrt{5} & 1 & 3/2+1/2\sqrt{5} \end{pmatrix}$$

and

$$C^{-1} = \begin{pmatrix} 1/50\left(5+3\sqrt{5}\right)\sqrt{5} & -1/50\left(5+\sqrt{5}\right)\sqrt{5} & 1/5 \\ -2/5 & -1/5 & 2/5 \\ 1/50\left(-5+3\sqrt{5}\right)\sqrt{5} & -1/50\left(-5+\sqrt{5}\right)\sqrt{5} & 1/5 \end{pmatrix}.$$

The vector of node counts is

$$N^{(d)} = \begin{pmatrix} 1/50\left(1-\sqrt{5}\right)^d\left(5+3\sqrt{5}\right)\sqrt{5}+2/5+ \\ 1/50\left(1+\sqrt{5}\right)^d\left(-5+3\sqrt{5}\right)\sqrt{5} \\ 1/50\left(1-\sqrt{5}\right)\left(1-\sqrt{5}\right)^d\left(5+3\sqrt{5}\right)\sqrt{5}+2/5+ \\ 1/50\left(1+\sqrt{5}\right)\left(1+\sqrt{5}\right)^d\left(-5+3\sqrt{5}\right)\sqrt{5} \\ 1/50\left(3/2-1/2\sqrt{5}\right)\left(1-\sqrt{5}\right)^d\left(5+3\sqrt{5}\right)\sqrt{5}-2/5+ \\ 1/50\left(3/2+1/2\sqrt{5}\right)\left(1+\sqrt{5}\right)^d\left(-5+3\sqrt{5}\right)\sqrt{5} \end{pmatrix}$$

such that the exact total number of nodes in depth $d$ is

$$1/50\left(7/2-3/2\sqrt{5}\right)\left(1-\sqrt{5}\right)^d\left(5+3\sqrt{5}\right)\sqrt{5}+2/5+$$
$$1/50\left(7/2+3/2\sqrt{5}\right)\left(1+\sqrt{5}\right)^d\left(-5+3\sqrt{5}\right)\sqrt{5}$$

The number of corner nodes (1, 0, 2, 2, 10, 26, 90, ...), the number of side nodes (0, 2, 2, 10, 26, 90, 282, ...) and the number of middle nodes (0, 0, 6, 22, 70, 230,...) grow as expected. The largest eigenvalue $1+\sqrt{5}$ dominates the growth of the search tree in the limit for large $d$.

In the Twenty-Four-Puzzle the value $N^{(d)}$ equals

$$\begin{pmatrix} 1/36\left(-2\sqrt{3}\right)^d+2/9\left(-\sqrt{3}\right)^d+2/9\left(\sqrt{3}\right)^d+1/36\left(2\sqrt{3}\right)^d \\ -1/18\sqrt{3}\left(-2\sqrt{3}\right)^d-2/9\sqrt{3}\left(-\sqrt{3}\right)^d+2/9\sqrt{3}\left(\sqrt{3}\right)^d+1/18\sqrt{3}\left(2\sqrt{3}\right)^d \\ 1/18\left(-2\sqrt{3}\right)^d+1/9\left(-\sqrt{3}\right)^d+1/9\left(\sqrt{3}\right)^d+1/18\left(2\sqrt{3}\right)^d \\ 1/12\left(-2\sqrt{3}\right)^d+1/12\left(2\sqrt{3}\right)^d \\ -1/18\sqrt{3}\left(-2\sqrt{3}\right)^d+1/9\sqrt{3}\left(-\sqrt{3}\right)^d-1/9\sqrt{3}\left(\sqrt{3}\right)^d+1/18\sqrt{3}\left(2\sqrt{3}\right)^d \\ 1/36\left(-2\sqrt{3}\right)^d-1/9\left(-\sqrt{3}\right)^d-1/9\left(\sqrt{3}\right)^d+1/36\left(2\sqrt{3}\right)^d \end{pmatrix}$$

for the following total of nodes in depth $d$

$$n^{(d)} = 1/36\ \left(7-4\sqrt{3}\right)\left(-2\sqrt{3}\right)^d + 1/9\ \left(2-\sqrt{3}\right)\left(-\sqrt{3}\right)^d +$$
$$1/9\ \left(2+\sqrt{3}\right)\left(\sqrt{3}\right)^d + 1/36\ \left(7+4\sqrt{3}\right)\left(2\sqrt{3}\right)^d.$$

The value for small $d$ validates that the total number of nodes increases as expected (2,6,18,60,198,684,...). Once again the vector of the largest absolute value determines the search tree growth.

If the size of the system is large, the exact value of $N^{(d)}$ has to be approximated. One option to bypass the intense calculations for determinants of large matrices and roots of high-degree polynomials is to compute the asymptotic branching factor $b$. The number of nodes in the brute-force search tree is then approximated by $n^{(d)} \approx b^d$.

## 4.6 Approximate Prediction of Search Tree Size

The matrix denotation for calculating the population of nodes according to the given equivalence relation implies $N^{(d)} = PN^{(d-1)}$, with $N^{(d)}$ being the vector of equivalent class sizes. The asymptotic branching factor $b$ is given by the limit of $||N^{(d)}||_1/||N^{(d-1)}||_1$ which equals $N_i^{(d)}/N_i^{(d-1)}$ in any component $i \in \{1,\ldots,k\}$. Evaluating $N_i^{(d)}/N_i^{(d-1)}$ for increasing depth $d$ is exactly what is considered in the algorithm of van Mises for approximating the largest eigenvalue (in absolute terms) of $P$. The algorithm is also referred to as the *power iteration* method.

As a precondition, the algorithm requires that $P$ be diagonizable. This implies that we have $n$ different eigenvalues $\lambda_1,\ldots,\lambda_n$ and each eigenvalue $\lambda_i$ with multiplicity of $\alpha_i$ has $\alpha_i$ linear independent eigenvectors. Without loss of generality, we assume that the eigenvalues are given in decreasing order $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_k|$. The algorithm further requires that the start vector $N^{(0)}$ have a representation in the basis of eigenvectors in which no coefficient according to $\lambda_1$ is trivial.

We distinguish the following two cases: $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_k|$ and $|\lambda_1| = |\lambda_2| > \ldots \geq |\lambda_k|$. In the first case we obtain that (independent of the choice of $j \in \{1,\ldots,k\}$) the value of $\lim_{d\to\infty} N_j^{(d)}/N_j^{(d-1)}$ equals $|\lambda_1|$. Similarly, in the second case $\lim_{d\to\infty} N_j^{(d)}/N_j^{(d-2)}$ is in fact $\lambda_1^2$. The cases $|\lambda_1| = \ldots = |\lambda_l| > \ldots \geq |\lambda_k|$ for $l > 2$ are dealt with analogously. The outcome of the algorithm and therefore the limit in the number of nodes in layers with difference $l$ is $|\lambda_1|^l$, so that once more the geometric mean turns out to be $|\lambda_1|$.

We indicate the proof of the first case only. Diagonizability implies a basis of eigenvectors $b_1\ldots,b_k$. Due to $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$ the quotient of $|\lambda_i/\lambda_1|^d$ converges to zero for large values of $d$. If the initial vector $N^{(0)}$ with respect to the eigenbasis is given as $x_1b_1 + x_2b_2 + \ldots + x_kb_k$ applying $P^d$ yields $x_1P^db_1 + x_2P^db_2 + \ldots + x_kP^db_k$ by linearity of $P$, which further reduces to $x_1b_1\lambda_1^d + \lambda_2^dx_2b_2 + \ldots + \lambda_n^dx_kb_k$ by the definition of eigenvalues and eigenvectors. The term $x_1b_1\lambda_1^d$ will dominate the sum for increasing values of $d$. Factorizing $\lambda_1^d$ in the numerator and $\lambda_1^{d-1}$ in denominator of the quotient of $N_j^{(d)}/N_j^{(d-1)}$ results in an equation of the form $x_1b_1\lambda_1 + R$ where $\lim_{d\to\infty} R$ is bounded by a constant, since except of the leading term $x_1b_1\lambda_1$ both numerator and denominator in $R$ only involve expressions of the form $O(|\lambda_i/\lambda_1|^d)$. Therefore, to find the asymptotic

branching factor analytically, it suffices to determine the set of eigenvalues of $P$ and to take the largest one. This corresponds to the results of the asymptotic branching factors in the $(n^2-1)$-Puzzles.

In the Eight-Puzzle the ratio $N_1^{(d)}/N_1^{(d-2)}$ is equal to 8 for $d > 3$ and, therefore, approximates $\lambda_1^2$. The value $n^{(d)}/\sqrt{8}^d$ alternates between $3/4$ and $1/\sqrt{2}$. Hence, $\sqrt{8}^d$ approximates the search tree growth.

For the Fifteen-Puzzle for increasing depth $d$ the value $N_1^{(d)}/N_1^{(d-1)}$ equals 1, 3, 13/5, 45/13, 47/15, 461/141, 1485/461, 4813/1485, 15565/4813, 50381/15565, 163021/50381, 527565/163021 = 3.236178161, etc., a sequence approximating $1+\sqrt{5} = 3.236067978$. Moreover, the ratio of $n^{(d)}$ and $(1+\sqrt{5})^d$ quickly converges to
$1/50\,\left(7/2+3/2\,\sqrt{5}\right)\left(-5+3\,\sqrt{5}\right)\sqrt{5} = .5236067984$.

In the Twenty-Four-Puzzle the ratio $N_1^{(d)}/N_1^{(d-2)}$ converges to 12 starting with the sequence 6, 9, 11, 129/11, 513/43, 683/57, 8193/683, 32769/2731, 43691/3641 = 11.99972535, etc. The quotient $n^{(d)}/\sqrt{12}^d$ for larger depth alternates between .3888888889 and .3849001795 and is therefore bounded by a small constant.

If $n$ is even – as in the Fifteen-Puzzle – the largest eigenvalue is unique and if $n$ is odd – as in the Eight- and in the Twenty-Four-Puzzle – we find two eigenvalues with the same absolute value verifying that every two depths the node sizes will asymptotically increase by the square of these values.

## 4.7 Generalizing the Approach

Iterating the algorithm with $||N^{(d)}||_1/||N^{(d-1)}||_1$ instead of $N_j^{(d)}/N_j^{(d-1)}$ shows that the convergence conditions according to $G$ and $\overline{G}$ are equivalent. This is important, since other graph properties may alter, e.g. symmetry of $A_G$ is not inherited by $A_{\overline{G}}$. Therefore, we concentrate on diagonizability results of $A_G$, which are easier to obtain. The *Theorem of Schur* states that symmetric matrices are indeed diagonizable. Moreover, the eigenvalues are real and the matrix to perform the basis transformation has the eigenvectors in its columns.

For the $(n^2-1)$-Puzzle we are done. Since $G$ is undirected, $A_G$ is indeed symmetric. In the spectrum of $A_G$ power iteration either obtains a unique branching factor $b = |\lambda_1|$ or a branching factor of $\lambda_1^2$ for every two iterations. Therefore, the branching factor is the *spectral radius* $\rho = |\lambda_1|$.

### 4.7.1 Other Problem Spaces

Since the search tree is often exponentially larger than the problem graph we have reduced the prediction of the search tree growth to the spectral analysis of the explicit adjacency representation of the graph. As long as this graph is available, accurate and approximate predictions for the brute-force and subsequently for the heuristic search tree growth can be computed.

However, the calculations for large implicitly given graphs are involved such that reduction of the analysis to a smaller structure is desirable. For the $(n^2-1)$-Puzzle we proposed a compression to a few branching classes. The application of equivalence class reduction to exactly predict the search tree growth relies on the regular structure of the

problem space. This technique is available as long as the same branching behavior for different states is given.

For *Rubik's Cube* without predecessor elimination $N^{(d)}$ equals $18^d$ since all nodes in the search tree span a complete 18-ary subtree. With predecessor elimination the node branching factor reduces to 15, since for each of the three twists *single clockwise*, *double clockwise*, and *counterclockwise* there is a remainder of five sides *front*, *back*, *right*, *left*, *up*, and *down* that are available. If we further restrict rotation of opposite sides to exactly one order we get the transition matrix $((6\ 6), (9\ 6))$, where the first class is the set of primary nodes with branching factor 15, and the second class is the class of secondary nodes with branching factor 12. The eigenvalues are $6 + 3\sqrt{6}$ and $6 - 3\sqrt{6}$ and the value $n^{(d)}$ equals $1/2\ \left(6 + 3\ \sqrt{6}\right)^d + 1/2\ \left(6 - 3\ \sqrt{6}\right)^d$. For small values of $d$ experimental data as given in [215] matches this analytical study. The observed asymptotic branching factor is $6 + 3\sqrt{6} = 13.34846923$ as expected.

Extending the work to problem domains like the PSPACE-complete *Sokoban* problem [74] is challenging. It is difficult to derive an accurate prediction, since the branching behavior of the tree includes almost all state facets. Therefore, a more complicated search model has to be devised to derive exact or approximate search tree prediction in this domain. As Andreas Junghanns has coined in his Ph.D. dissertation [199], the impact of the search tree node prediction formula $\sum_{d=0}^{c} n^{(d)} P(c - d)$ has still to be shown. In the other PSPACE-complete sliding block game *Atomix* [194, 184] simplification based on branching equivalences do apply and yield savings that are exponential in the number of atoms, but this void labeling scheme still results in an intractable size of the equivalence graph structure. Only very small games can be analyzed by this method.

### 4.7.2 Pruning

When incorporating pruning to the exploration process, symmetry of the underlying graph structure may be affected. Once again we consider the Eight-Puzzle. The adjacency matrix $A_G^{pred}$ for predecessor elimination now consists of four classes: $cs$, $sc$, $mc$ and $cm$, where the class $ij$ indicates that the predecessor of a $j$-node in the search tree is an $i$ node.

$$A_G^{pred} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

In this case we cannot infer diagonizability according to the set of real numbers. Fortunately, we know that the branching factor is a positive real value since the iteration process is real. Therefore, we may perform all calculation to predict the search tree growth with complex numbers, for which the characteristic polynomial factorizes. The branching factor and the search tree growth can be calculated analytically and the iteration process eventually converges. In the example, the set of (complex) eigenvalues is $i\sqrt{2}$, $-i\sqrt{2}$, $\sqrt{3}$, and $-\sqrt{3}$. Therefore, the asymptotic branching factor is $\sqrt{3}$. The vector $N^{(d)}$ equals

$$\begin{pmatrix} 1/5\,\left(i\sqrt{2}\right)^d + 1/5\,\left(-i\sqrt{2}\right)^d + 3/10\,\left(\sqrt{3}\right)^d + 3/10\,\left(-\sqrt{3}\right)^d \\ -1/10\,i\sqrt{2}\left(i\sqrt{2}\right)^d + 1/10\,i\sqrt{2}\left(-i\sqrt{2}\right)^d + 1/10\,\sqrt{3}\left(\sqrt{3}\right)^d - 1/10\,\sqrt{3}\left(-\sqrt{3}\right)^d \\ 3/20\,i\sqrt{2}\left(i\sqrt{2}\right)^d - 3/20\,i\sqrt{2}\left(-i\sqrt{2}\right)^d + 1/10\,\sqrt{3}\left(\sqrt{3}\right)^d - 1/10\,\sqrt{3}\left(-\sqrt{3}\right)^d \\ -1/10\,\left(i\sqrt{2}\right)^d - 1/10\,\left(-i\sqrt{2}\right)^d + 1/10\,\left(\sqrt{3}\right)^d + 1/10\,\left(-\sqrt{3}\right)^d \end{pmatrix}.$$

Finally, the total number of nodes in depth $d$ is

$$n^{(d)} = 1/5\,\left(1/2 + 1/4\,i\sqrt{2}\right)\left(i\sqrt{2}\right)^d + 1/5\,\left(1/2 - 1/4\,i\sqrt{2}\right)\left(-i\sqrt{2}\right)^d +$$

$$1/10\,\left(4 + 2\,\sqrt{3}\right)\left(\sqrt{3}\right)^d + 1/10\,\left(4 - 2\,\sqrt{3}\right)\left(-\sqrt{3}\right)^d.$$

For small values of $d$ the value $n^{(d)}$ equals 1, 2, 4, 8, 10, 20, 34, 68, 94, 188 etc.

### 4.7.3 Non-Diagonizability

Since we assumed diagonizability, the eigenspaces $L(\lambda_i)$ according to the values $\lambda_i$ have full rank $\alpha_i$. In general this is not true. Not all matrices are diagonalizable. In this case the best thing one can do is to transform the matrix into *Jordan Form* which has blocks on the diagonal, each block being $r \times r$, with the eigenvalue on the diagonal, 1's above the diagonal and 0's everywhere else. More precisely, a matrix $A$ has *Jordan Form* $J$ for an invertible matrix $T$, if $J = T^{-1}AT$ consists of so-called Jordan-blocks $J_1, \ldots, J_m$. One Jordan-block has an eigenvalue on the main diagonal and 1s on the diagonal above. Therefore, $T$ gives a basis of eigenvectors and so-called *main vectors*. Each Jordan-block $J_l$ of dimension $j_l$ corresponds to one eigenvector $t_1$ and $j_l - 1$ main vectors $t_2, \ldots, t_{j_l}$ with $(A - \lambda_i I)t_0 = 0$ and $(A - \lambda_i I)t_m = t_{m-1}$, $m = 2, \ldots, j_l$. Using the Jordan basis one can devise $P^d N^{(0)}$ similar to the case above.

### 4.7.4 Start Vector

The second subtlety arises even if the matrix is diagonalizable. We are interested in determining the behavior of $P^d N^{(0)}$ for large $d$, where $P$ is an $n \times n$ matrix and $N^{(0)}$ is an $n \times 1$ vector. Suppose that $P$ is diagonalizable, which means that there is a basis of eigenvectors. Hence, $N^{(0)}$ can be written as a sum of eigenvectors: $N^{(0)} = v_1 + v_2 + v_3 + \ldots + v_n$ where $v_i$ is an eigenvector with eigenvalue $\lambda_i$. It follows that $P^d N^{(0)} = \lambda_1^d v_1 + \lambda_2^d v_2 + \lambda_3^d v_3 + \ldots + \lambda_n^d v_n$ So the term with the largest corresponding eigenvalue will dominate for large $d$, *provided that the eigenvector is non-zero*. It may happen that the initial vector $v$ has component of zero in the eigenspace of the largest eigenvalue. In general, the algorithm finds the largest eigenvalue in which the corresponding component is non-zero. Fortunately, this observation is more theoretical in nature. In the iteration process this case is very rarely fulfilled. Rounding errors will soon or later lead to non-zero components. To determine the asymptotic branching factor we have several initial states to choose from such that at least one has to yield non-zero coefficients.

## 4.8 Previous Work

This paper extends the work of Edelkamp and Korf [104] that already derived the asymptotic branching factors of the sliding-tile puzzles and Rubik's Cube. However, their approach lack sufficient convergence conditions. We established the criterion of diagonizability of the adjacency graph matrix of the problem graph that emerges of the algorithm of van Mises and showed that this criterion is fulfilled in undirected graphs by the Theorem of Schur. The $(n^2 - 1)$-Puzzles and Rubik's Cube are chosen to illustrate the techniques, since they are inherently difficult to solve and often considered in case studies.

The set of recurrence relations in [104] also showed that the numbers of nodes at various depths can be calculated in time linear to the product of the number of node classes and the depth of search tree by numerically iterating the recurrence relations. In contrast to this finding, the current paper resolves the problem of how to compute a closed form for the number of nodes. Last but not least, the given mathematical formalization of equivalence classification, diagonalization and power iteration builds a bridge for more powerful results in applying known mathematical theorems. At least in theory, generality to different problem spaces is given, since this approach applies to any problem graph with a diagonizable matrix and probably to more than that.

## 4.9 Conclusion and Discussion

In the paper we have improved the prediction of the number of node expansions in IDA* by an exact derivation of the number of nodes in the brute-force search tree. We have resolved the question of convergence to explain anomalies in of the asymptotic branching factor. The asymptotic branching factor is the spectral radius of the successor generation matrix and can be computed with the power iteration method. The approach extends to further regular problem spaces and can cope with simple pruning rules. The main result is that diagonizability is granted in undirected problem graphs, such that exact and approximate calculation of the brute-force search-tree are mathematically sound. The technique for establishing a closed form is not standard, and it is hard to suggest other methodologies to actually solve the set of recurrence relations.

Moreover, given the adjacency matrix $P$ of an undirected graph studuing $N_i^{(d)}/N_i^{(d-1)}$ and $N_i^{(d)}/N_i^{(d-k)}$, $k > 1$ of the equation $N^{(d)} = N^{(d-1)}P$ gives the (mean) asymptotic branching factor. This is in fact the algorithm of van Mises to determine the largest eigenvalue of $P$ for whose applicability we have to test if $P$ is diagonizable. The paper closes the small gap in literature to accurately predict search tree growth in closed form and to compute the branching factor both analytically and numerically without relying on strong experimental assumption on the convergence.

Since for practical problems in which IDA* applies it is very unlikely that the entire graph structure can be kept in main memory, the approach helps only if some reduction of the branching behavior with respect to equivalence classes can be obtained. Therefore, the analysis is limited to the cases where the the successor generator matrix of the original or the adjacency graph structure can be build. If not, abstractions to the graph structure have to be found that preserve or approximate information of the branching behavior.

All analyses given in this or precursory papers on search tree prediction do not include the application of transposition tables, in which visited states together with their best

encountered state merits (path length plus heuristic estimate) are kept. This in fact is also a challenge for analysts. One option is the prediction of the search tree growth of IDA* with respect to bit-state hashing, which turns out to be an improvement to transposition tables in single-agent games [194] and protocol verification [109]. For this model of partial search first results on coverage prediction have been found [111].

Exact calculation of the brute-force search tree raises the question if the other source of uncertainty, namely the heuristic equilibrium distribution, can also be eliminated. As said, the equilibrium distribution of the estimate can be obtained by random sampling. However, in some cases of regular search trees exact values can be produced. If the estimate is given with respect to a pattern database storing pairs of the form (estimated value, state pattern) by analyzing the pattern database, a histogram of heuristic values can computed: we determine the number of states that satisfy a pattern with a total to be computed for each integral heuristic value in a predefined range. For consistent heuristics this range will be bounded by the heuristic estimate of the start state and the optimal solution length. At the very far end of this research line there are precise or approximate predictions for the growth of A*'s and IDA*'s search efforts according to various kinds of heuristics, node caching strategies and problem domains. This implies an alternative way of defining heuristics themselves: ranking successor nodes according to the expected growth of the resulting search tree.

# Part II

# Puzzle Solving

# Paper 5

# Finding Optimal Solutions to Atomix

Stefan Edelkamp
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

Falk Hüffner, Henning Fernau, and Rolf Niedermeier.
Wilhelm-Schickard Institut für Informatik,
Universität Tübingen,
Sand 13, D-72076 Tübingen,
eMail: {hueffner,fernau,niedermr}@informatik.uni-tuebingen.de

In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science. Springer, pp. 229-243, 2001.

**Abstract**

We present solutions of benchmark instances to the solitaire computer game Atomix found with different heuristic search methods. The problem is PSPACE-complete. An implementation of the heuristic algorithm A* is presented that needs no priority queue, thereby having very low memory overhead. The limited memory algorithm IDA* is handicapped by the fact that, due to move transpositions, duplicates appear very frequently in the problem space; several schemes of using memory to mitigate this weakness are explored, among those, "partial" schemes which trade memory savings for a small probability of not finding an optimal solution. Even though the underlying search graph is directed, backward search is shown to be viable, since the branching factor can be proven to be the same as for forward search.

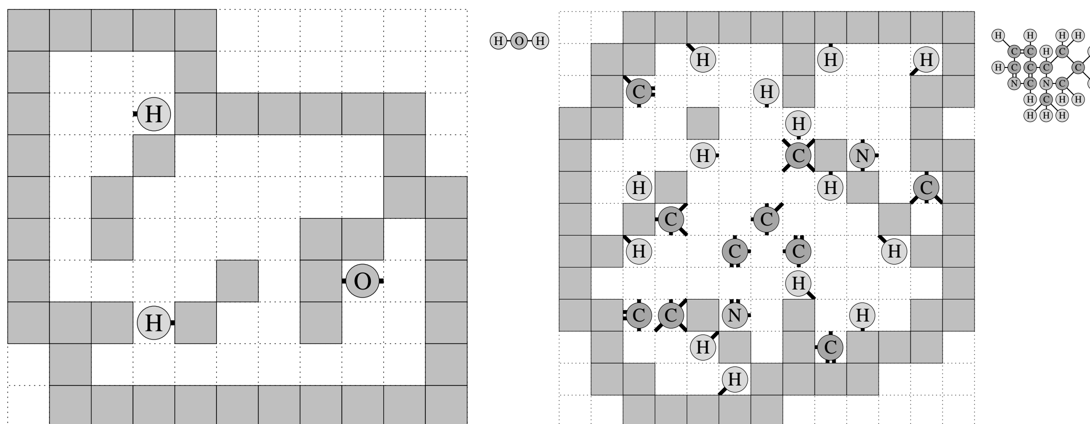

Figure 5.1: Two Atomix problems. The left one—which is Atomix 01 in the list of the appendix—can be solved with the following 13 moves, where the atoms are numbered left-to-right in the molecule: 1 down left, 3 left down right up right down left down right, 2 down, 1 right. The right one—which is level number 43 from the "katomic" implementation—illustrates a more complex problem; it takes at least 66 moves to solve.

## 5.1 Introduction

*Atomix* was invented in 1990 by Günter Krämer and first published by Thalion Software for the popular computer systems of that time. The goal is to assemble a given molecule from atoms (see Fig. 5.1). The player can select an atom at a time and "push" it towards one of the four directions north, south, west, and east; it will keep on moving until it hits an obstacle or another atom. The game is won when the atoms form the same constellation (the "molecule") as depicted beside the board. A concrete Atomix problem, given by the original atom positions and the goal molecule, is called a *level* of Atomix.

The original game had a time limit and did not count the moves needed; we will instead focus on the analytical aspect and try to minimize the solution length as a goal. Note that we are only interested in *optimal* solutions; in order to just find any solution fast, quite different algorithms would be necessary.

An implementation of this Atomix variation for the X Window System is available as "katomic" from `http://games.kde.org`. A JavaScript version can be played online at `http://www.sect.mce.hw.ac.uk/˜peteri/atomix`.

Our solver program written in C++ is able to solve 17 of the 30 problems from the original Atomix and 18 of the 67 problems from katomic optimally. In an appendix, we list a selection of these findings.

## 5.2 Heuristic Search

Many common problems and, especially, most solitaire puzzles can be formulated as a *state space search* problem: given are a start state, a set of goal states and a set of operators to transform one state into another; wanted is a sequence of operators, also simply called a *move sequence*, that transforms the start state into a goal state and that is of minimal length. A state space can be represented as a graph, with nodes representing

| | 24-Puzzle | Sokoban | Atomix |
|---|---|---|---|
| Branching factor | 2–4 | 0–50 | 12–40 |
| effective | 2.3 | 10 | 7 |
| Solution Length | 80–112 | 97–674 | 8–120 |
| typical | 100 | 260 | 45 |
| Search space size | $10^{25}$ | $10^{18}$ | $10^{21}$ |
| Graph | Undirected | Directed | Directed |

Table 5.1: Search space properties of some puzzles. The effective branching factor is the number of children of a state, after applying memory-bounded pruning methods (in particular, not utilizing transposition tables; see Sect. 5.5.3 for the methods applied to Atomix). For Sokoban and Atomix, the numbers are for typical puzzles from the human-made test sets; for Sokoban, those problems are about $20 \times 20$ and, for Atomix, about $16 \times 16$ squares large.

states and (directed) edges representing moves. That way, well-known graph algorithms can be applied. To emphasize this aspect, states generated in a state space search are often called "nodes".

For hard combinatorial problems, the use of *heuristics* can often lead to dramatic improvements for a state space search. Many problems would even be unsolvable without them. For a state space search, "heuristic" has a well-defined meaning: an estimate of the moves left from the current state to a goal. Of special interest are *admissible* heuristics: they never overestimate the number of moves. The well-known algorithms A* and IDA* can be proven to always find an optimal solution when using an admissible heuristic. An admissible heuristic judges the "quality" of a state $s$: if $g(s)$ is the number of moves already applied, and $h(s)$ is the heuristic estimate, then $f(s) := g(s) + h(s)$ is a lower bound on the total number of moves. This number, customarily called the "$f$-value", can be used in two ways: to guide the search and to reduce the effective depth of the search. The first idea naturally leads to the A* algorithm: "promising" states are examined first. The second is applied in the IDA* algorithm: "hopeless" states are not examined at all.

## 5.3 Related Puzzles

The following table compares some search space properties of Atomix to other games. The results are contained in [104, 199, 221].

Due to its close relationships to Atomix (which will become important in the next section), we discuss the 15- and the 24-puzzle as special instances of the $(n^2 - 1)$-puzzle in more details.

The 15-puzzle consists of a square tray of size $4 \times 4$ with 15 tiles numbered $1$ through $15$ and one empty square. A move consists of sliding one tile adjacent to the empty square into the empty space. The goal is to obtain the usual ordering of the numbers on the tiles by some move sequence. The 15-puzzle is likely to be the most thoroughly analyzed puzzle of this kind [221]. It serves as a kind of "fruit fly" for heuristic search. It is easy to implement, has an obvious heuristic with the "Manhattan distance", and not too large

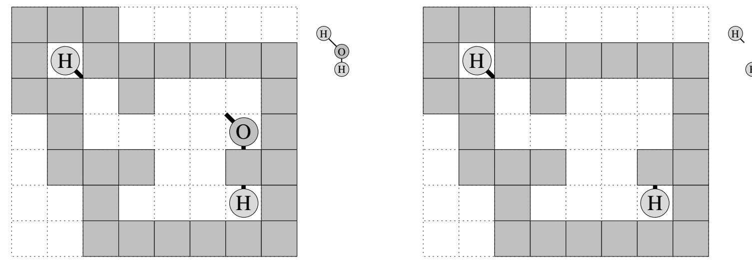


Figure 5.2: The left problem can be solved in 13 moves. We cannot get a lower bound by leaving out one atom, as in the right picture; the problem even becomes unsolvable.

a search space. The Manhattan distance heuristic can be calculated by summing up the number of turns it would take for a tile to get to its goal position if it was the only tile in the tray. This is obviously a lower bound on the actual number of turns.

Many search methods developed for the 15-puzzle can be easily adapted for Atomix. One important difference is that the underlying search graph for Atomix is directed; not every move can be undone.

Improved heuristics for the 15-puzzle make it possible to solve even the extended "24-puzzle"-variation [221]. Most of them follow the common theme of examining a subproblem where only a few tiles are regarded and most are ignored. The "linear conflict heuristic" [159], for example, tries to find pairs of tiles in a row or column which need to pass each other to get to the goal position. In such a case, another two moves can be added to the heuristic given by the Manhattan distance, since one tile will have to move out of the way and back. The work of Culberson and Schaeffer [75] generalizes this idea to "pattern databases": Each possible distribution of the tiles 1–8 on the board is analyzed and solved, yielding a lower bound which is often better than the Manhattan heuristic with the linear conflict heuristic, since there are more tile interactions. The same is done for the other 7 tiles. Unfortunately, these powerful techniques cannot be directly applied to Atomix, since removing atoms from a state does not necessarily make it easier to solve; in fact, it can even become unsolvable, see Fig. 5.2.

## 5.4 Complexity of Atomix

### 5.4.1 Complexity of Sliding-Block Puzzles

The time complexity of sliding block puzzles was the subject of intense research in the past. Though seemingly trivial, most variations are at least NP-hard and, some, even PSPACE-complete. The following table shows some results. The table was basically taken from Demaine et al. [77], extended by the category of games where the blocks are pushed by an external agent not represented on the board, into which Atomix falls. The columns mean:

1. Are the moves performed by a robot on the board, or by an outside agent?
2. Can the robot pull as well as push?
3. Does each block occupy a unit square, or may there be larger blocks?

4. Are there fixed blocks, or are all blocks movable?

5. How many blocks can be pushed at a time?

6. Does it suffice to move the robot/a special block to a certain target location, instead of pushing *all* blocks into their goal locations?

7. Will the blocks "keep sliding" when pushed until they hit an obstacle?

8. The dimension of the puzzle: is it 2D or 3D?

| Game | 1. Robot | 2. Pull | 3. Blocks | 4. Fixed | 5. # | 6. Path | 7. Slide | 8. Dim. | 9. Complexity |
|---|---|---|---|---|---|---|---|---|---|
| PushPush3D | + | − | unit | − | 1 | + | + | 3D | NP-hard |
| PushPush | + | − | unit | − | 1 | + | + | 2D | NP-hard |
| Push-$*$ | + | − | unit | − | $k$ | − | − | 2D | NP-hard |
| Sokoban+ | + | − | 1×2 | + | 2 | − | − | 2D | PSPACE-compl. |
| Sokoban | + | − | unit | + | 1 | − | − | 2D | PSPACE-compl. [74] |
| 15-Puzzle | − | | unit | − | 1 | − | − | 2D | NP-compl. [301] |
| Rush Hour | − | | 1×{2,3} | − | 1 | + | − | 2D | PSPACE-compl. |
| Atomix | − | | unit | + | 1 | − | + | 2D | PSPACE-compl. [184] |

### 5.4.2 A Formal Definition of Atomix

We will now give a formal definition of an Atomix problem instance (*level*).

**Definition 1** *An Atomix problem instance consists of:*

- *A finite set $A$ of so-called* atom types.

- *A game* board $B = \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$.

- *A bit matrix $O = (O[p] \in \{0,1\} \mid p \in B)$ of size $w \times h$ (the* obstacles*). A* position *is simply a tuple $p = (p_x, p_y) \in B$. A* state *$s$ is defined as a subset of $A \times B$. An element of $s$ is also called an* atom*. Note that the same atom type might appear several times in a state.*

  *A position $p = (p_x, p_y)$ is said to be* empty *for a state $s$ if $O[p] = 0$ and there is no $a \in A$ with $(a, (p_x, p_y)) \in s$.*

  *Positions outside of $B$ areassumed not to be empty.*

- *A state $S$ (the* start state*), which satisfies that, for all $(a, p) \in S$, $O[p] = 0$.*

- *A state $G$ (the* goal state*). For the problem to be solvable, for all $(a, p) \in G$, $O[p] = 0$ and there must be a bijection between $S$ and $G$ where each atom in $S$ maps onto an atom in $G$ with the same atom type.*

*A* direction $(d_x, d_y)$ *is a tuple of $x$ and $y$ offsets, i. e., one of* $(0, -1)$, $(1, 0)$, $(0, 1)$ *and* $(-1, 0)$. *A* move *is a tuple of a position $p$ and a direction $d$. For a state $s$, a move $(p, d)$ is only legal if there is an atom $(a, p)$ in $s$, and $(p_x + d_x, p_y + d_y)$ is empty.*

*Applying a move $(p, d)$ to a state $s$ will yield another state $s'$ in which every atom has the same position, except the atom $(a, p)$: it will be replaced by $(a, p')$ with $p' = (p_x + \delta d_x, p_y + \delta d_y)$, where $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \leq \delta$, and $(p_x + (\delta + 1)d_x, p_y + (\delta + 1)d_y)$ is not empty. A* solution *is a sequence of moves which, incrementally applied to the start state, yields the goal state.*

The main difference between this formal definition and the informal introduction is that the goal positions of the atoms are given explicitly. The reason is that this makes the puzzle both easier to analyze and to implement. Since the number of goal positions is linear in the board size, this difference does not affect the time complexity significantly. Our implementation handles different possible goal positions by imposing a move limit and trying all possible goal positions with that limit, and repeating with an incremented move limit until a solution is found.[1]

### 5.4.3 The Hardness of Atomix

**Proposition 1** *Atomix on an $n \times n$ board is NP-hard.*

**Proof:** Any $(n^2 - 1)$-puzzle instance can be transformed into an Atomix instance by replacing the numbered tiles with atoms of unique atom types. For the $(n^2 - 1)$-puzzle, a legal move consists of sliding a tile into the empty space. In the reduction, those are also the only legal moves, since all atoms not adjacent to the empty square cannot satisfy the move legality condition, and those adjacent to the empty square can only take its place as a move. As shown by Ratner and Warmuth, the $(n^2 - 1)$-puzzle is NP-complete [301], so Atomix is NP-hard. ∎

**Proposition 2** *Atomix on an $n \times n$ board is in PSPACE.*

**Proof:** A nondeterministic Turing-machine can solve Atomix by repeatedly applying a legal move from the start state encoded on its tape until a goal is reached. The number of possible Atomix states is limited by $n^2!$; hence, the machine can announce that the puzzle is unsolvable after having applied more moves without finding a solution. Since an encoding of an Atomix state needs only polynomial space, it follows that Atomix is in NPSPACE = PSPACE. ∎

Very recently, Holzer and Schwoon [184] showed by reduction from *non-empty intersection of finite automata* that Atomix is even PSPACE-complete. They also provide a level with an exponentially long optimal solution.

## 5.5 Searching the State Space of Atomix

Much progress has been made in the area of heuristic search. This is due to: faster machines with more memory, better heuristics, and better search methods. Of these three, by far, the largest improvements come from better heuristics.

[1] As explained later, this incremental approach is already inherent to IDA*, and can be applied to A* with reasonable overhead.

### 5.5.1 Heuristics for Atomix

As is often the case, a heuristic for Atomix can be devised by examining a model with relaxed restrictions. We drop the condition that an atom slides as far as possible: it may stop at any closer position. These moves are called *generalized moves*.[2] In order to obtain an easily computable heuristic, we also allow that an atom may also pass through other atoms or share a place with another atom. The goal distance in this model can be summed up for all atoms to yield an admissible heuristic for the original problem.

The following properties are immediate consequences of the definition.

**Property 1** *The heuristic is admissible.*

**Property 2** *The $h$-values of child states can only differ from that of the parent state by $0$, $+1$ or $-1$.*

**Property 3** *The heuristic is* monotone *(consistent) , i. e., the $f$-value of a child state cannot be lower than the $f$-value of the parent state.*

Apart from this somewhat obvious heuristic, it proved to be pretty hard to make any improvements. Two ideas were considered, but not implemented due to their limited applicability:

If an atom needs a "stopper" at a certain position to make a turn for each optimal path, but no optimal path of any atom has an intermediate position at the stopper position, $h$ can be incremented by one.

If an atom is alone in a "cave", for some positions, one or two moves can be added to the heuristic (see the example below). A "cave" is an area that contains no goal position and has only one entry; if an atom is alone in there, it cannot use any stoppers unless another atom leaves its optimal path. This heuristic has a greater potential, since it can be added up admissibly for each cave. Unfortunately, only a few levels from our test set contain caves which could yield improved heuristics.

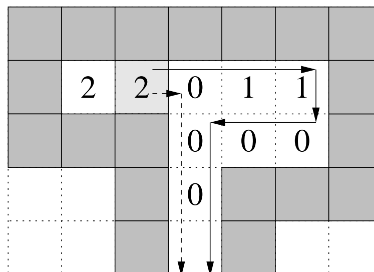


Figure 5.3: An example for the "cave"-heuristic: if only one atom is in the cave, the number denoted on its square can be added to the heuristic estimate. For example, an atom on the light grey square has to take the path marked with a solid line, instead of the optimal path of generalized moves marked with a dashed line, which is two moves shorter.

[2]The variant of Atomix which uses generalized moves has an undirected search graph. Atomix with generalized moves on an $n \times n$ board is also NP-hard but is in PSPACE.

### 5.5.2 A*

A* is one of the oldest heuristic search algorithms [161]. It is very time-efficient, but needs an exponential amount of memory. A* remembers all states ever encountered, which is the reason for its exponential space complexity. A priority queue holds all states that have not yet been expanded. It is sorted by the $f$-value of the states. Nodes are popped from the queue and expanded afterwards. The children are inserted into the queue or discarded if they were already encountered. Sometimes, the same state is reached with a lower $g$-value; in that case, its entry in the state table has to be updated and it will be re-inserted into the queue. With an admissible heuristic, A* will always find an optimal solution.

The state table is usually implemented as a hash table for fast access and low memory overhead. The priority queue can be implemented with a bucket for each $f$-value, containing all open states with that $f$-value. In Sect. 5.6.2, we present an alternative implementation that only needs the state table and does without a priority queue.

### 5.5.3 IDA*

Iterative Deepening A* (IDA*) (see [213]) was the first algorithm that allowed finding optimal solutions to the 15-puzzle. IDA* performs a series of depth-first searches, with an increasing move limit. The heuristic is used to prune subtrees where it is known that the bound will be exceeded, since the $f$-value is larger than the bound. Each iteration will visit all nodes encountered in the previous iteration again; but, since the majority of nodes will be generated in the last iteration, this does not affect the time complexity.

IDA* uses no memory except for the stack, so its memory use is linear in the search depth. Also, since it needs no intricate data structures, it can be implemented very efficiently. But of course, this comes at a price: IDA* does not detect *transpositions* in the search graph. If a state is encountered that has already been expanded and dismissed, it will be expanded again, possibly resulting in the re-evaluation of a huge subtree. There are two approaches to lessen this weakness: use of problem specific knowledge and use of memory.

**Pruning the Search Space.** Several techniques are known for pruning, e. g., predecessor elimination, which disallows to take back moves immediately. For games with undirected underlying graphs like the 15-puzzle, this is an obvious optimization. For Atomix, it can still be applied, since pushing an atom into the opposite direction immediately after a move always yields the same state as pushing it in that direction in the first place.

**Move Pruning.** When examining a solution move sequence for an Atomix level, one notices that many, though not all moves could be interchanged. Interchanging moves is not possible in four cases, as is explained in Fig. 5.4.

The idea is to check if a generated move is independent of the previous move (i. e., applying them in reversed order would yield the same state) and, if they are independent, to impose an arbitrary order (the atom with the lower number must move first). This scheme has proven to be very efficient in avoiding transpositions, reducing running time by several orders of magnitudes.

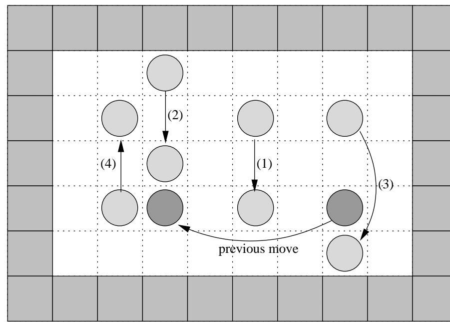


Figure 5.4: There are four cases where two moves are dependent, i. e., their order cannot be interchanged: (1) The current atom would have stopped the previously moved atom earlier. (2) The current atom uses the previously moved atom as a stopper. (3) The current atom would stop earlier if the previously moved atom had not been moved. (4) The current atom was the stopper of the previously moved atom.

### 5.5.4 Partial IDA*

Analogously to the two-player game search, a *transposition table* can be used to avoid re-expanding states [305]. States are inserted into a hash table together with their $g$-value as they are generated. Then, for each newly generated state, it is looked up whether it has already been expanded with the same or a lower $g$-value so it can be pruned. If memory was unlimited, this would avoid all possible transpositions. Many schemes have been proposed for proper management of the transposition table with limited memory [90]; our implementation simply refuses to insert states into an exhausted table.

A lot of memory can be saved with *Partial IDA** [109, 111]. This idea originates in the field of *protocol verification*, where the objective is to generate all reachable states and check if they fulfill a certain criterion. A hash table is used to avoid re-expanding states. Just as for a single-agent search, memory is the limiting resource. Therefore, Holzmann suggested *bit-state hashing* [188]: instead of storing the complete state, only a single bit corresponding to the hash value is set, indicating that this state has been visited before. Because of the possibility of hash collisions, states might get pruned erroneously, so this method can give false positives. When applied to IDA*, states on optimal paths could get pruned, so the method looses admissibility, but is still useful to determine upper bounds and likely lower bounds.

For Atomix, initial experiments with Partial IDA* rarely found optimal solutions. The reason is that just knowing a state has been encountered before is not sufficient, because if we encounter it with a lower $g$-value than previously, it needs to be expanded again. To achieve this, we include $g$ into the hash value and look up with $g$ and $g - 1$. This means transpositions with better $g$ will not be found in the table and expanded, as desired. Transpositions with $g$ worse by 2 or more will also not be detected; experiments showed that they are rare and the resulting subtrees are shallow, though.

By probing twice (with $g$ and $g - 1$), we increase the likelihood of hash collisions. For example, if we declare the table to be full if every 8th bit is set, we have an effective memory usage of 1 byte per state, and a collision probability of $1 - \left(\frac{7}{8}\right)^2 = 23\%$. To improve the collision resistance, one can calculate a second hash value and always set and check two bits, effectively doubling memory usage but lowering collision probability

to $1 - \left(\frac{63}{64}\right)^2 = 3\%$.

A related scheme with better memory efficiency and collision resistance is *hash compaction* [350]. It utilizes a hash table where, instead of the complete state, only a hash signature is saved. In our implementation, we use 1 byte for the signature, and probe for $g$ and $g-1$. This way, we have a collision probability of $1 - \left(\frac{255}{256}\right)^2 = 0.8\%$, so even if there is only a single possible solution of length 30, the probability of finding it is $\left(\left(\frac{255}{256}\right)^2\right)^{30} = 79\%$; and in fact, all 47 solutions found this way were optimal.

Different policies are possible in the case of a hash collision detected by differing signatures. Usual hash table techniques like chaining or open addressing can be applied. We tried a much simpler scheme: the old entry gets overwritten. This can be seen as a special case of the *$t$-limited scheme* proposed by Stern and Dill [331] with $t = 1$. One disadvantage of this scheme is that entries will already get overwritten before the table is completely full. Since for the "interesting" (difficult) cases, the state table will fill up soon anyway, this effect is limited.

### 5.5.5 Backward Search

Many puzzles are *symmetric*, i. e., the set of children of a state equals the set of possible parents. This is equivalent to the state space graph being undirected. As already mentioned, this is the case for the 15-puzzle, but not for Sokoban or Atomix. For Atomix, it is simple to find all potential parent states, though: they can be found by applying all legal *backward moves*. In a backward move, an atom being pushed may stop moving at any position, but it can only be pushed in a direction if it is adjacent to an obstacle in the *opposite* direction.

Formally defined, a backward move is a triple of a position $p$, a direction $d$, and a distance $\delta$. It is legal for a state $s$ if there is an atom $(a, p)$ in $s$, and $(p_x - d_x, p_y - d_y)$ is *not* empty, and $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \leq \delta$. Applying a backward move is analogous to applying a forward move.

Expanding states for backward Atomix is about as easy as for forward Atomix, and the same heuristic can be used, since the generalized moves from Sect. 5.5.1 comprise backward moves. Hence, the crucial point is the branching factor.

**Lemma 1** *The sum of possible forward moves and the sum of possible backward moves of all states of a level are identical and, therefore, the average number of children for backwards expansion is exactly the same as for forward expansion.*

**Proof:** We first show the equality for a single atom by structural induction. On a board with no empty squares, the equation is trivially true. We show it also remains true when removing an obstacle. The change in the number of moves depends on the pattern of empty squares around the obstacle being removed; we examine all possible patterns (up to symmetry, and omitting the trivial case of 4 obstacles), as illustrated in Fig. 5.5, with $a, b, c$ and $d$ being the number of empty squares in each direction.

(a) 3 adjacent obstacles: $1 - b + b + 1 = 1 + 1 = 2$.

(b) 2 adjacent obstacles, where the obstacles are diagonally adjacent:
$1 - b + b + d + 2 - d + 1 = 1 + 2 + 1 = 4$.

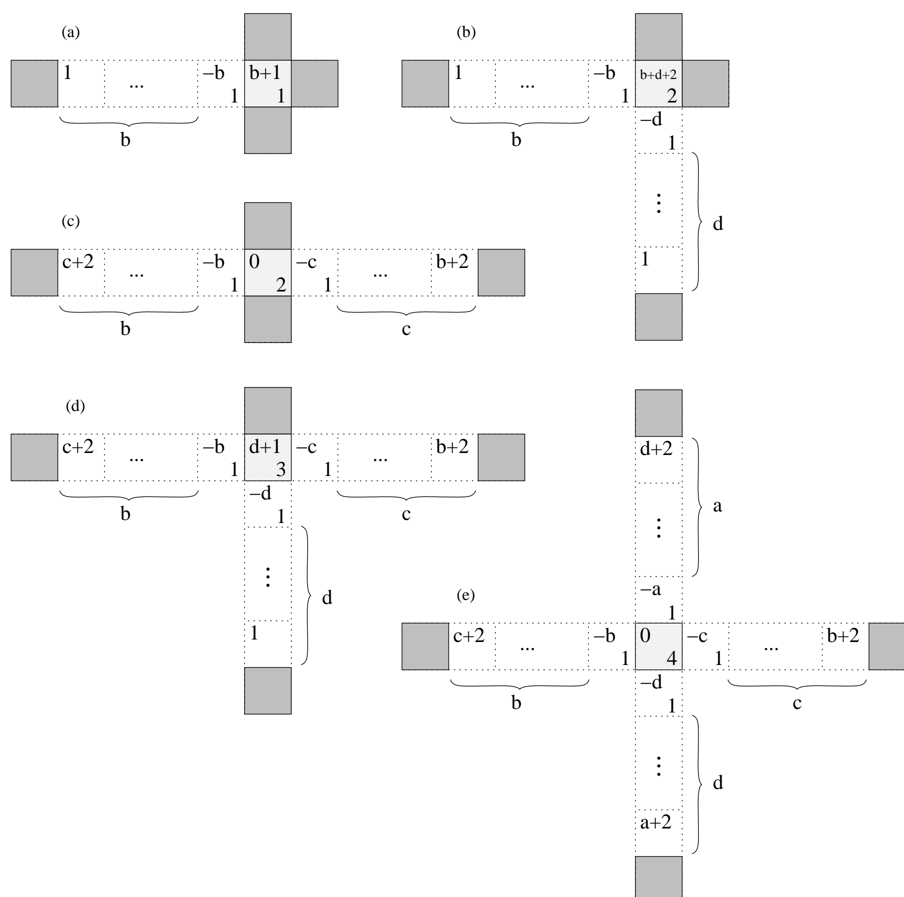


Figure 5.5: The light grey obstacle in the center is being removed. The upper left corner of each square denotes the number of backward moves that are lost or gained by this change for an atom on this square. The lower right corner denotes the number of new forward moves. Squares which are skipped in the sketches (denoted by dots) have zero gain with respect to both forward and backward moves.

(c) 2 adjacent obstacles, where the obstacles are opposite:
$c + 2 - b + 0 - c + b + 2 = 1 + 2 + 1 = 4.$

(d) 1 adjacent obstacle: $c + 2 - b + d + 1 - c + b + 2 - d + 1 = 1 + 3 + 1 + 1 = 6.$

(e) no adjacent obstacles:
$d + 2 - a + c + 2 - b + 0 - c + b + 2 - d + a + 2 = 1 + 1 + 4 + 1 + 1 = 8.$

Now, let us consider the contribution of one atom to the possible moves. Each possible distribution of the other atoms can be considered as a pattern of obstacles. With the observation just made, the sum of possible forward and backward moves is the same when summing up over all possible positions of the considered atom; so the sum over all possible distributions of the other atoms is also identical and, since this equality holds for each atom, the lemma is true.

■

In practice, the branching factors can differ substantially, since the generated states are not random; the move operators make certain states more likely than others, and states close to the goal where (by convention) all atoms are close together are much more likely. In our experiments, we observed differences up to 30% in forward and backward branching factors.

## 5.6 Implementation

### 5.6.1 Identical Atoms

The presence of undistinguishable atoms (i. e., atoms with identical atom types) poses problems for an implementation: The heuristic cannot simply perform a table lookup to find a lower bound for an atom, since it is not clear which atom should go to which goal position. To find a good lower bound, a *minimum cost perfect matching* has to be done for each set of identical atoms to find the cheapest assignment of atoms to goal positions. Minimum cost perfect matching for a bipartite graph can be solved using minimum cost augmentation in time quadratic in the number of identical atoms [226].

### 5.6.2 A*

An implementation of A* needs the following operations: check if a state has been encountered before and with which $g$-value, find an open state with optimal $f$-value, mark an open state as closed, and update the $g$-value of a saved state to a lower value.

This is usually implemented with a hash table and a priority queue which stores all open states. We will show that if the heuristic is monotone, no priority queue is actually needed: an optimal open state can be found efficiently without any additional data structures. Our algorithm is easy to implement and time and space efficient.

Initially, the available memory is allocated for two tables: the *state table* and the *hash table*. As states are generated, they are appended to the end of the state table; states never get deleted. The states are tagged with an *open*-bit and with the $g$-value. The hash table stores a pointer into the state table at the position corresponding to the hash value of the state; this allows a quick lookup of states. A linear displacement scheme is used to

resolve hash collisions. The monotonicity of the heuristic implies that $f_{\text{opt}}$, the currently optimal $f$-value of an open state, is also monotone over the run of A*. To find an optimal open state, a linear search on the state table is performed until an open state with $f = f_{\text{opt}}$ is found. The following proposition shows that this can be done efficiently:

**Proposition 3** *In A* with a monotone heuristic with a hash table and no additional data structure, a state with optimal $f$-value can be found in amortized time O(branching factor).*

**Proof:** To achieve this, we need to ensure that, for each $f_{\text{opt}}$-value, when we reach the end of the state table, we have expanded all states with $f = f_{\text{opt}}$, so we don't have to go through the table again. This can be ensured by not upgrading a state in place if it is re-encountered with lower $g$, but to append it at the end like new states. States with $f < f_{\text{opt}}$ will never be reopenened [115], so this suffices to ensure the desired property.

Two kinds of states will be skipped because their $f$-value differs from $f_{\text{opt}}$:

- Closed states with $f < f_{\text{opt}}$. We keep a pointer to the very first open state, so only closed states with $f = f_{\text{opt}} - 1$ or $f = f_{\text{opt}} - 2$ have to be skipped; for any branching factor greater than 1, this can be at most twice as many as states with $f = f_{\text{opt}}$ and, with a higher branching factor, their number even becomes negligible.

- Open states with $f > f_{\text{opt}}$. They must have been generated by states with $f = f_{\text{opt}}$ or $f = f_{\text{opt}} - 1$, so their number is linear in the number of states with $f = f_{\text{opt}}$ and the branching factor.

■

Our implementation with this scheme is several times faster than a naïve implementation using the C++ STL `priority_queue` and `set`, which are based on heaps, resp., binary trees, with a memory overhead of about 30 bytes per state. On a Pentium III with 500 MHz, it can generate around a million states per second.

A disadvantage of this scheme is that it is not possible to further discriminate among optimal states. A common idea to speed up A* is to sort among states with equal $f$-values those closer to the top that are further advanced.

To trade time for memory, the A* implementation works iteratively: similarly to IDA*, an artificial upper bound on the number of moves is applied and, if the $f$-value of a generated state exceeds this bound, it is pruned. If then the search fails, it is restarted with the bound increased by one. This also allows us to take multiple goal positions into account. Due to the exponential behavior, this slows down the search only by a constant factor.

## 5.7 Conclusions

Atomix proved itself to be a challenging puzzle; this is corroborated by the recent PSPACE-completeness proof. The classic algorithms A* and IDA* have been implemented and adapted to the problem domain; we have found optimal solutions for many problems from our benchmark set. Our A* implementation with a single data structure for the *open* and *closed* set can solve “smaller” puzzles very efficiently. With Partial IDA*

based on hash compaction, we have presented a memory-bounded scheme that makes excellent use of the available memory and has low runtime overhead; improved bounds on the error probability would be useful, though. Further progress is likely to come from improved heuristics rather than from better search methods, since our current heuristic is rather uninformed. We have shown that while the search graph is directed, the backward branching factor does not differ from the forward branching factor; this makes Atomix an interesting testbed for bidirectional algorithms.

## 5.8 Experimental Results

The experiments were performed on a Pentium III with 500 MHz, utilizing 128 MB of main memory and imposing a time limit of one hour. The source can be found at `http://www-fs.informatik.uni-tuebingen.de/~hueffner`.

| Level | Atoms | Goals | Man | A* | IDA* | IDA*-tt | IDA*-r | PIDA* |
|---|---|---|---|---|---|---|---|---|
| Atomix 01 | 3 | 17 | | $= 13$ | $= 13$ | $= 13$ | $= 13$ | $= 13$ |
| Atomix 02 | 5 | 6 | | $= 21$ | $= 21$ | $= 21$ | $= 21$ | $= 21$ |
| Atomix 03 | 6 | 4 | | $= 16$ | $= 16$ | $= 16$ | $= 16$ | $= 16$ |
| Atomix 04 | 6 | 2 | | $\geq 23$ | $\geq 22$ | $= 23$ | $= 23$ | $= 23$ |
| Atomix 05 | 9 | 2 | | $\geq 34$ | $\geq 34$ | $\geq 35$ | $\geq 35$ | $\geq 37$ |
| Atomix 06 | 8 | 4 | | $= 13$ | $= 13$ | $= 13$ | $= 13$ | $= 13$ |
| Atomix 07 | 9 | 1 | | $\geq 25$ | $\geq 26$ | $= 27$ | $\geq 25$ | $= 27$ |
| Atomix 09 | 7 | 1 | | $= 20$ | $= 20$ | $= 20$ | $= 20$ | $= 20$ |
| Atomix 10 | 10 | 2 | | $\geq 28$ | $\geq 28$ | $\geq 28$ | $\geq 27$ | $\geq 30$ |
| Atomix 11 | 5 | 14 | | $= 14$ | $= 14$ | $= 14$ | $= 14$ | $= 14$ |
| Atomix 12 | 9 | 4 | | $= 14$ | $= 14$ | $= 14$ | $= 14$ | $= 14$ |
| Atomix 13 | 8 | 1 | | $= 28$ | $= 28$ | $= 28$ | $= 28$ | $= 28$ |
| Atomix 15 | 12 | 1 | | $\geq 35$ | $\geq 36$ | $\geq 37$ | $\geq 37$ | $\geq 37$ |
| Atomix 16 | 9 | 2 | | $\geq 26$ | $\geq 26$ | $\geq 27$ | $\geq 25$ | $\geq 28$ |
| Atomix 18 | 8 | 4 | | $= 13$ | $= 13$ | $= 13$ | $= 13$ | $= 13$ |
| Atomix 22 | 8 | 3 | | $\geq 24$ | $\geq 24$ | $\geq 25$ | $\geq 23$ | $\geq 27$ |
| Atomix 23 | 4 | 20 | | $= 10$ | $= 10$ | $= 10$ | $= 10$ | $= 10$ |
| Atomix 26 | 4 | 17 | | $= 14$ | $= 14$ | $= 14$ | $= 14$ | $= 14$ |
| Atomix 28 | 10 | 1 | | $\geq 28$ | $\geq 29$ | $\geq 29$ | $\geq 26$ | $\geq 29$ |
| Atomix 29 | 8 | 2 | | $= 22$ | $= 22$ | $= 22$ | $= 22$ | $= 22$ |
| Atomix 30 | 8 | 4 | | $= 13$ | $= 13$ | $= 13$ | $= 13$ | $= 13$ |
| Unitopia 01 | 3 | 41 | 11 | $= 11$ | $= 11$ | $= 11$ | $= 11$ | $= 11$ |
| Unitopia 02 | 4 | 5 | 22 | $= 22$ | $= 22$ | $= 22$ | $= 22$ | $= 22$ |
| Unitopia 03 | 5 | 12 | 16 | $= 16$ | $= 16$ | $= 16$ | $= 16$ | $= 16$ |
| Unitopia 04 | 6 | 5 | 20 | $= 20$ | $= 20$ | $= 20$ | $= 20$ | $= 20$ |
| Unitopia 05 | 6 | 7 | 21 | $= 20$ | $= 20$ | $= 20$ | $= 20$ | $= 20$ |
| Unitopia 06 | 9 | 2 | 33 | $\geq 29$ | $\geq 30$ | $\geq 30$ | $\geq 30$ | $\geq 31$ |
| Unitopia 07 | 10 | 1 | 36 | $\geq 33$ | $\geq 33$ | $\geq 34$ | $\geq 32$ | $\geq 35$ |
| Unitopia 08 | 7 | 4 | 25 | $= 23$ | $= 23$ | $= 23$ | $= 23$ | $= 23$ |
| Unitopia 10 | 8 | 2 | 41 | $\geq 36$ | $\geq 36$ | $\geq 37$ | $\geq 38$ | $\geq 40$ |

| | |
|---|---|
| Man | Best result found by participants of an online game |
| IDA*-tt | IDA* with transposition table |
| IDA*-r | IDA* backward search with transposition table |
| PIDA* | Partial IDA* with hash compaction to 1 byte |

**Time performance.** A* runs out of memory usually much before a runtime of one hour and, so, can establish less stringent bounds. The advantage of using a transposition table for IDA* outweighs its runtime overhead and yields better results in all cases. Reverse search performs similar to forward search, as founded by the theroretical findings. Partial IDA* consistently beats IDA* with conventional hash tables because of better memory utilization and less runtime overhead. Note that most of these differences are expected to be more significant if the time limit is increased.

# Part III

# Action Planning

# Paper 6

# Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length

Stefan Edelkamp and Malte Helmert.
Albert-Ludwigs-Universität,
Am Flughafen 17, D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science, Springer, pages 135–147, 1999.

**Abstract**

In this paper we present a general-purposed algorithm for transforming a planning problem specified in Strips into a concise state description for single state or symbolic exploration.

The process of finding a state description consists of four phases. In the first phase we symbolically analyze the domain specification to determine constant and one-way predicates, i.e. predicates that remain unchanged by all operators or toggle in only one direction, respectively.

In the second phase we symbolically merge predicates which lead to a drastic reduction of state encoding size, while in the third phase we constrain the domains of the predicates to be considered by enumerating the operators of the planning problem. The fourth phase combines the result of the previous phases.

## 6.1 Introduction

Single-state space search has a long tradition in AI. We distinguish memory sensitive search algorithms like A* [161] that store the explored subgraph of the search space and approaches like IDA* and DFBnB [213] that consume linear space with respect to the search depth. Especially on current machines memory sensitive algorithms exhaust main memory within a very short time.

On the other hand linear search algorithms explore the search tree of generating paths, which might be exponentially larger than the underlying problem graph. Several techniques such as transposition tables [305], finite state machine pruning [337], and heuristic pattern databases [75] have been proposed. They respectively store a considerable part of the search space, exhibit the regular structure of the search problems, and improve the lower bound of the search by retrieving solutions to problem relaxations. Last but not least, in the last decade several memory restricted algorithms have been proposed [90, 311]. All memory restricted search approaches cache states at the limit of main memory.

Since finite state machine pruning is applicable only to a very restricted class of symmetric problems, single-state space search algorithms store millions of fully or partially defined states. Finding a good compression of state space is crucial. The first step is to efficiently encode each state; if we are facing millions of states we are better off with a small state description length.

The encoding length is measured in bits. For example one instance to the well-known Fifteen Puzzle can be compressed to 64 bits, 4 bits for each tile.

Single-state algorithms have been successful in solving "well-informed domains", i.e. problems with a fairly elaborated lower bound [199, 215], for good estimates lead to smaller parts of the search tree to be considered. In solving one specific problem, manually encoding the state space representations can be devised to the user. In case of AI planning, however, we are dealing with a family of very different domains, merely sharing the same, very general specification language. Therefore planners have to be general-purposed. Domain-dependent knowledge has either to be omitted or to be inferred by the machine.

Planning domains usually have large branching factors, with the branching factor being defined as the average number of successors of a state within planning space. Due to the resulting huge search spaces planning resists almost all approaches of single-state space search. As indicated above automated finite state pruning is generally not available although there is some promising research on symmetry leading to good results in at least some domains [130].

On the other hand, domain-independent heuristic guidance in form of a lower bound can be devised, e.g. by counting the number of facts missing from the goal state. However, these heuristics are too weak to regain tractability. Moreover, new theoretical results in heuristic single-state search prove that while finite state machine pruning can effectively reduce the branching factor, in the limit heuristics cannot [104, 219]. The influence of lower bounds on the solution length can best be thought of as a decrease in search depth. Therefore, even when incorporated with lower bound information, the problem of large branching factors when applying single-state space searching algorithms to planning domains remains unsolved. As a solution we propose a promising symbolic search technique also favoring a small binary encoding length.

## 6.2 Symbolic Exploration

Driven by the success of model checking in exploring search spaces of $10^{20}$ states and beyond, the new trend in search is reachability analysis [259]. Symbolic exploration bypasses the typical exponential growth of the search tree in many applications. However, the length of the state description severely influences the execution time of the relevant algorithms. In symbolic exploration the rule of thumb for tractability is to choose encodings of not much more than 100 bits.

Edelkamp and Reffel have shown that and how symbolic exploration leads to promising results in solving current challenges to single-agent search such as the Fifteen Puzzle and Sokoban [112]. Recent results show that these methods contribute substantial improvements to deterministic planning [113].

The idea in symbolically representing a set $S$ is to devise a boolean function $\phi_S$ with input variables corresponding to bits in the state description that evaluates to true if and only if the input $a$ is the encoding of one element $s$ in $S$. The drawback of choosing boolean formulae to describe $\phi_S$ is that satisfiability checking is NP-complete. The unique representation with binary decision diagrams (BDDs) can grow exponentially in size, but, fortunately, this characteristic seldom appears in practice [50].

BDDs allow to efficiently encode sets of states. For example let $\{0, 1, 2, 3\}$ be the set of states encoded by their binary value. The characteristic function of a single state is the minterm of the encoding, e.g. $\phi_{\{0\}}(x) = \overline{x_1} \wedge \overline{x_2}$. The resulting *BDD* has two inner nodes. The crucial observation is that the *BDD* representation of $S$ increases by far slower than $|S|$. For example the *BDD* for $\phi_{\{0,1\}} = \overline{x_1}$ consists of one internal node and the *BDD* for $\phi_{\{0,1,2,3\}}$ is given by the 1-sink only.

An operator can also been seen as an encoding of a set. In contrast to the previous situation a member of the transition relation corresponds to a pair of states $(s', s)$ if $s'$ is a predecessor of $s$. Subsequently, the transition relation $T$ evaluates to 1 if and only if $s'$ is a predecessor of $s$. Enumerating the cross product of the entire state space is by far too expensive. Fortunately, we can set up $T$ symbolically by defining which variables change due to an operator and which variables do not.

Let $S_i$ be the set of states reachable from the start state in $i$ steps, initialized by $S_0 = \{s\}$. The following equation determines $\phi_{S_i}$ given both $\phi_{S_{i-1}}$ and the transition relation: $\phi_{S_i}(s) = \exists s' \; (\phi_{S_{i-1}}(s') \wedge T(s', s))$. In other words we perform breadth first search with *BDDs*. A state s belongs to $S_i$ if it has a predecessor in the set $S_{i-1}$ and there exists an operator which transforms $s'$ into $s$. Note that on the right hand side of the equation $\phi$ depends on $s'$ compared to $s$ on the left hand side. Thus, it is necessary to substitute $s$ with $s'$ in the *BDD* for $\phi_{S_i}$. Fortunately, this substitution corresponds to a simple renaming of the variables.

Therefore, the key operation in the exploration is the *relational product* $\exists v (f \wedge g)$ of a variable vector $v$ and two boolean functions $f$ and $g$. Since existential quantification of one boolean variable $x_i$ in the boolean function $f$ is equal to disjunction $f|_{x_i=0} \vee f|_{x_i=1}$, the quantification of $v$ results in a sequence of subproblem disjunctions. Although computing the relational product is NP-hard in general, specialized algorithms have been developed leading to an efficient determination for many practical applications.

In order to terminate the search we test, if a state is contained in the intersection of the symbolic representation of the set $S_i$ and the set of goal states $G$. This is achieved by evaluating the relational product *goalReached* $= \exists x \; (\phi_{S_i} \wedge \phi_G)$. Since we enumerated

$S_0, \ldots, S_{i-1}$ in case *goalReached* evaluates to 1, $i$ is known to be the optimal solution length.

## 6.3 Parsing

We evaluate our algorithms on the AIPS'98 planning contest problems[1], mostly given in Strips [126]. An operator in Strips consists of pre- and postconditions. The latter, so-called effects, divide into an add list and a delete list.

Extending Strips leads to ADL with first order specification of conditional effects [289] and PDDL, a layered planning description domain language. Although symbolic exploration and the translation process described in this paper are not restricted to Strips, for the ease of presentation we will keep this focus.

A PDDL-given problem consists of two parts. In the domain specific part, predicates and actions are defined. A predicate is given by its name and its parameters, and actions are given by their names, parameters, preconditions, and effects. One example domain, Logistics, is given as follows[2].

```
(define (domain logistics-strips)
  (:predicates (OBJ ?obj) (TRUCK ?tru) (LOCATION ?loc)
               (AIRPLANE ?plane) (CITY ?city) (AIRPORT ?airport)
               (at ?obj ?loc)
               (in ?obj ?obj)
               (in-city ?obj ?city))

(:action LOAD-TRUCK
  :parameters (?obj ?tru ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?tru) (LOCATION ?loc)
                     (at ?tru ?loc) (at ?obj ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?tru)))

(:action UNLOAD-TRUCK
  :parameters (?obj ?tru ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?tru) (LOCATION ?loc)
                     (at ?tru ?loc) (in ?obj ?tru))
  :effect (and (not (in ?obj ?tru)) (at ?obj ?loc)))

(:action DRIVE-TRUCK
  :parameters (?tru ?loc-from ?loc-to ?city)
  :precondition  (and (TRUCK ?tru) (LOCATION ?loc-from)
                      (LOCATION ?loc-to) (CITY ?city)
                      (at ?tru ?loc-from)
                      (in-city ?loc-from ?city)
                      (in-city ?loc-to ?city))
  :effect (and (not (at ?tru ?loc-from)) (at ?tru ?loc-to)))
...
)
```

[1] http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html.

[2] Dots (...) are printed if source fragments are omitted.

The problem specific part defines the objects to be dealt with and describes initial and goal states, consisting of a list of facts (instantiations to predicates).

```
(define (problem strips-log-x-1)
   (:domain logistics-strips)
   (:objects package6 package5 package4 package3 package2 package1
             city6 city5 city4 city3 city2 city1
             truck6 truck5 truck4 truck3 truck2 truck1
             plane2 plane1
             city6-1 city5-1 city4-1 city3-1 city2-1 city1-1
             city6-2 city5-2 city4-2 city3-2 city2-2 city1-2)
   (:init (and (OBJ package1) (OBJ package2)
               ...
               (at package6 city3-1)
               ... )
   )
   (:goal (and (at package6 city1-2)
               ... )
   )
)
```

Using current software development tools, parsing a PDDL specification is easy. In our case we applied the Unix programs *flex* and *bison* for lexically analyzing the input and parsing the result into data structures. We used the standard template library, STL for short, for handling the different structures conveniently. The information is parsed into vectors of predicates, actions and objects. All of them can be addressed by their name or a unique numeric identifier, with STL maps allowing conversions from the former ones to the latter ones. Having set up the data structures, we are ready to start analyzing the problem.

## 6.4 Constant and One-Way Predicates

A *constant predicate* is defined as a predicate whose instantiations are not affected by any operator in the domain. Since Strips does not support types, constant predicates are often used for labeling different kinds of objects, as is the case for the `TRUCK` predicate in the Logistics domain. Another use of constant predicates is to provide persistent links between objects, e.g. the `in-city` predicate in the Logistics domain which associates locations with cities. Obviously, constant predicates can be omitted in any state encoding.

Instantiations of *one-way* predicates do change over time, but only in one direction. There are no one-way predicates in the Logistics domain; for an example consider the Grid domain, where doors can be opened with a key and not be closed again. Thus `locked` and `open` are both one-way predicates. Those predicates need to be encoded only for those objects that are not listed as open in the initial state. In PDDL neither constant nor one-way predicates are marked and thus both have to be inferred by an algorithm. We iterate on all actions, keeping track of all predicates appearing in any effect lists. Constant predicates are those that appear in none of those lists, one-way predicates are those that appear either as add effects or as delete effects, but not both.

## 6.5 Merging Predicates

Consider the Logistics problem given above, which serves as an example for the remaining phases. There are 32 objects, six packages, six trucks, two airplanes, six cities, six airports, and six other locations. A naive state encoding would use a single bit for each possible fact, leading to a space requirement of $6 \cdot 32 + 3 \cdot 32^2 = 3264$ bits per state, since we have to encode six unary and three binary predicates. Having detected constant predicates, we only need to encode the `at` and `in` predicates, thus using only $2 \cdot 32^2 = 2048$ bits per state. Although this value is obviously better, it is far from being satisfying.

A human reader will certainly notice that it is not necessary to consider all instantiations of the `at` predicate independently. If a given package $p$ is at location $a$, it cannot be at another location $b$ at the same time. Thus it is sufficient to encode *where* $p$ is located, i.e. we only need to store an object number which takes only $\lceil \log 32 \rceil = 5$ bits per package. How can such information be deduced from the domain specification? To tell the truth, this is not possible, since the information does not only depend on the operators themselves but also on the initial state of the problem. If the initial state included the facts (`at` $p$ $a$) as well as (`at` $p$ $b$), then $p$ could be at multiple locations at the same time.

However, we can try to prove that the number of locations a given object is at cannot increase over time. For a given state, we define $\#at_2(p)$ as the number of objects $q$ for which the fact (`at` $p$ $q$) is true. If there is no operator that can increase this value, then $\#at_2(p)$ is limited by its initial value, i.e. by the number of corresponding facts in the initial state. In this case we say that `at` is *balanced* in the second parameter. Note that this definition can be generalized for $n$-ary predicates, defining $\#pred_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$ as the number of objects $p_i$ for which the fact (`pred` $p_1$ ... $p_n$) is true. If we knew that `at` was balanced in the second parameter, we would be facing one of the following situations:

- $\#at_2(p) = 0$: We have to store no information about the location of $p$.
- $\#at_2(p) = 1$: The location of $p$ can be encoded by using an object index, i.e. we need $\lceil \log o \rceil$ bits, where $o$ denotes the number of objects $p$ can be assigned to in our problem.
- $\#at_2(p) > 1$: In this case, we stick to naive encoding.

So can we prove that the balance requirement for `at` is fulfilled? Unfortunately we cannot, since there are some operators that increase $\#at_2$, namely the `UNLOAD-TRUCK` operator. However, we note that whenever $\#at_2(p)$ increases, $\#in_2(p)$ decreases, and vice versa. If we were to merge `at` and `in` into a new predicate `at+in`, this predicate would be balanced, since $\#(at + in)_2 = \#at_2 + \#in_2$ remains invariant no matter what operator is applied.

We now want to outline the algorithm for checking the balance of $\#pred_i$ for a given predicate $pred$ and parameter $i$: For each action $a$ and each of its add effects $e$, we check if $e$ is referring to predicate `pred`. If so, we look for a corresponding delete effect, i.e. a delete effect with predicate `pred` and the same argument list as $e$, except for the $i$-th argument which is allowed to be (and normally will be) different. If we find such a delete effect, it balances the add effect, and there is no need to worry.

If there is no corresponding delete effect, we search the delete effect list for any effect with a matching argument list (again, we ignore parameter $i$), no matter what predicate

it is referring to. If we do not find such an effect, our balance check fails. If we do find one, referring to predicate `other`, then we recursively call our algorithm with the merged predicate `pred+other`. Note that "matching argument list" does not necessarily mean that `other` takes its arguments in the same order as `pred`, which makes the actual implementation somewhat more complicated.

It is even possible to match `other` if that predicate takes one parameter less than `pred`, since parameter $i$ does not need to be matched. This is a special case in which $\#other_i(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$ can either be 1 or 0, depending on whether the corresponding fact is true or not, since there is no parameter $p_i$ here. Examples of this situation can be found in the Gripper domain, where `carry ?ball ?gripper` can be merged with `free ?gripper`.

If there are several candidates for `other`, all of them are checked, maybe proving balance of different sets of merged predicates. In this case, all of them are returned by the algorithm. It is of course possible that more than two predicates are merged in order to satisfy a balance requirement since there can be multiple levels of recursion. This algorithm checks the $i$-th parameter of predicate `pred`. Executing it for all predicates in our domain and all possible values of $i$ and collecting the results yields an exhaustive list of balanced merged predicates.

In the case of the Logistics domain, our algorithm exhibits that merging `at` and `in` gives us the predicate `at+in` which is balanced in the second parameter. Looking at the initial facts stated in the problem specification, we see that we can store the locations of trucks, airplanes and packages by using six bits each, since $\#(at + in)_2$ evaluates to one for those objects, and that we do not need to encode anything else, since the other objects start off with a count of zero.

Note that $\lceil \log 32 \rceil = 5$ bits are not sufficient for encoding locations at our current level of information, since we not only have to store the index of the object we are referring to, but also which of the two predicates `at` or `in` is actually meant. Thus our encoding size can be reduced to $(6 + 6 + 2) \cdot 6 = 84$ bits, which is already a reasonable result and sufficient for many purposes.

## 6.6 Exploring Predicate Space

However, we can do better. In most cases it is not necessary to allow the full range of objects for the balanced predicates we have detected, since e.g. a package can only be at a location or in a vehicle (truck or airplane), but never at another package, in a location, and so on.

If a fact is present in the initial state or can be instantiated by any valid sequence of operators, we call it *reachable*, otherwise it is called *unreachable*.

Many facts can be proven to be unreachable directly from the operators themselves, since actions like `LOAD-TRUCK` require the object the package is put into to be a truck. However, there are some kinds of unreachable facts we do not want to miss that cannot be spotted that way.

For example, `DRIVE-TRUCK` can only move a truck between locations in the same city, since for a truck to move from $a$ to $b$, there must be a city $c$, so that (`in-city` $a$ $c$) and (`in-city` $b$ $c$) are true. Belonging to the same city is no concept that is modeled directly in our Strips definition.

For those reasons, we do not restrict our analysis to the domain specification and instead take the entire problem specification into account. What we want to do is an exploration of predicate space, i.e. we try to enumerate all instantiations of predicates that are reachable by beginning with the initial set of facts and extending it in a kind of breadth-first search.

Note that we are exploring predicate space, not search space. We do not store any kind of state information, and only keep track of which facts we consider reachable. Thus, our algorithm can do one-side errors, i.e. consider a fact reachable although it is not, because we do not pay attention to mutual exclusion of preconditions. If a fact $f$ can be reached by an operator with preconditions $g$ and $h$, and we already consider $g$ and $h$ reachable, then $f$ is considered reachable, although it might be the case that $g$ and $h$ can never be instantiated at the same time. This is a price we have to pay and are willing to pay for reasons of efficiency. Anyway, if we were able to decide reliably if a given combination of facts could be instantiated at the same time, there would hardly remain any planning problem to be solved. We tested two different algorithms for exploring predicate space, *Action-Based Exploration* and *Fact-Based Exploration*.

## 6.6.1 Action-Based Exploration

In the action-centered approach, the set of reachable facts is initialized with the facts denoted by the initial state. We then instantiate all operators whose preconditions can be satisfied by only using facts that we have marked as reachable, marking new facts as reachable according to the add effect lists of the instantiated operators. We then again instantiate all operators according to the extended set of reachable facts. This process is iterated until no further facts are marked, at which time we know that there are no more reachable facts.

Our implementation of the algorithm is somewhat more tricky than it might seem, since we do not want to enumerate all possible argument lists for the operators we are instantiating, which might take far too long for difficult problems (there are e.g. $84^7 \approx 3 \cdot 10^{13}$ different possible instantiations for the `drink` operator in problem Mprime-14 from the AIPS'98 competition).

To overcome this problem, we apply a backtracking technique, extending the list of arguments one at a time and immediately checking if there is an unsatisfied precondition, in which case we do not try to add another argument. E.g., considering the `LOAD-TRUCK` operator, it is no use to go on searching for valid instantiations if the `?obj` parameter has been assigned an object $o$ for which (`OBJ` $o$) has not been marked as reachable.

There is a second important optimization to be applied here: Due to the knowledge we already have accumulated, we know that `OBJ` is a constant predicate and thus there is no need to dynamically check if a given object satisfies this predicate. This can be calculated beforehand, as well as other preconditions referring to constant predicates.

So what we do is to statically constrain the domains of the operator parameters by using our knowledge about constant and one-way predicates. For each parameter, we pre-compute which objects possibly could make the corresponding preconditions true. When instantiating operators later, we only pick parameters from those sets. Note that due to this pre-computation we do not have to check preconditions concerning constant unary predicates at all during the actual instantiation phase.

For one-way predicates, we are also able to constrain the domains of the corresponding

parameters, although we cannot be as restrictive as in the case of constant predicates. E.g., in the Grid example mentioned above, there is no use in trying to open doors that are already open in the initial state. However, we cannot make any assumption about doors that are closed initially.

There are two drawbacks of the action-based algorithm. Firstly, the same instantiations of actions tend to be checked multiple times. If an operator is being instantiated with a given parameter list, it will be instantiated again in all further iterations. Secondly, after a few iterations, changes to the set of reachable facts tend to become smaller and less frequent, but even if only a single fact is added to the set during an iteration, we have to evaluate all actions again, which is bad. Small changes should have less drastic consequences.

### 6.6.2 Fact-Based Exploration

Therefore we shift the focus of the exploration phase from actions to facts. Our second algorithm makes use of a queue in which all facts that are scheduled to be inserted into the set of reachable facts are stored. Initially, this queue consists of the facts in the initial state, while the set of reachable facts is empty. We then repeatedly remove the first fact $f$ from the queue, add it to the set of reachable facts and instantiate all operators whose preconditions are a subset of our set of reachable facts *and include* $f$. Add effects of these operators that are not yet stored in either the set of reachable facts or the fact queue are added to the back of the fact queue. This process is iterated until the fact queue is empty. Although it does not look as if much was gained at first glance, this algorithm is a big improvement to the first one.

The key difference is that when instantiating actions, only those instantiations need to be checked for which $f$ is one of the preconditions, which means that we can *bind* all parameters appearing in that precondition to the corresponding values of $f$, thus reducing the number of degrees of freedom of the argument lists. Of course, the backtracking and constraining techniques mentioned above apply here as well. The problem of multiple operator instantiations does not arise. We never instantiate an operator that has been instantiated with the same parameter list before, since we require $f$ to be one of the preconditions, and in previous iterations of the loop, $f$ was not regarded a reachable fact.

Returning to our Logistics problem, we now know that a package can only be at a location, in a truck or in an airplane. An airplane can only be at an airport, and a truck can only be at a location which must be in the same city as the location the truck started at.

## 6.7 Combining Balancing and Exploration

All we need to do in order to receive the encoding we are aiming at is to combine the results of the previous two phases. Note that these results are very different: While predicate space exploration yields information about the facts themselves, balanced predicates state information about the *relationship* between different facts. Both phases are independent of each other, and to minimize our state encoding, we need to combine the results.

In our example, this is simple. We have but one predicate to encode, the `at+in` predicate created in the merge phase. This leads to an encoding of 42 bits (cf. Table 6.1),

```
(5 bits) package6
  at  city6-1 city5-1 city4-1 city3-1 city2-1 city1-1
      city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
  in  truck6 truck5 truck4 truck3 truck2 truck1
      plane2 plane1
...
(5 bits) package1
  at  city6-1 city5-1 city4-1 city3-1 city2-1 city1-1
      city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
  in  truck6 truck5 truck4 truck3 truck2 truck1
      plane2 plane1
(1 bit) truck6
  at  city6-1 city6-2
...
(1 bit) truck1
  at  city1-1 city1-2
(3 bits) plane2
  at  city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
(3 bits) plane2
  at  city6-2 city5-2 city4-2 city3-2 city2-2 city1-2
```

Table 6.1: Encoding the Logistics problem 1-01 with 42 bits.

which is the output of our algorithm. However, there are cases in which it is not obvious how the problem should be encoded. In the Gripper domain (constant predicates omitted) the merge step returns the balanced predicates `at-robby`, `carry+free`, and `at+carry`; `at-robby` is an original operator, while `carry+free` and `at+carry` have been merged.

We do not need to encode each of the merged predicates, since this would mean encoding `carry` twice. If we had already encoded `carry+free` and now wanted to encode the `at+carry` predicate for a given object $x$, with $n$ facts of the type $(\texttt{at}\ x\ y)$ and $m$ facts of the type $(\texttt{carry}\ x\ y)$, we would only need $\lceil \log(n+1) \rceil$ bits for storing the information, since we only have to know which of the `at`-facts is true, or if there is no such fact. In the latter case, we know that some fact of the type $(\texttt{carry}\ x\ y)$ is involved and can look up which one it is in the encoding of `carry+free`. However, encoding `at+carry` first, thus reducing the space needed by `carry+free` is another possibility for encoding states, and is in fact the better alternative in this case. Since we cannot know which encoding yields the best results, we try them out systematically.

Although there is no need for using heuristics here since the number of conflicting possibilities is generally very small, we want to mention that as a rule of thumb it is generally a good idea to encode predicates that cover the largest number of facts first.

Predicates that are neither constant nor appear in any of the balanced merge predicates are encoded naively, using one bit for each possible fact. Those predicates are rare. In fact, in the considered benchmark set we only encountered them in the Grid domain, and there only for encoding the `locked` state of doors which obviously cannot further be compressed.

## 6.8 Experimental Results

In this section we provide data on the achieved compression to the state descriptions of the AIPS'98 planning competition problems. The problem suite consists of six different Strips domains, namely *Movie*, *Gripper*, *Logistics*, *Mystery*, *Mprime*, and *Grid*. In Table 6.1 we have exemplarily given the full state description for the first problem in the Logistics suite. The exhibited knowledge in the encoding can be easily extracted in form of *state invariants*, e.g. a package is either a location in a truck or in an airplane, each truck is restricted to exactly one city, and airplanes operate on airports only.

Table 6.2 depicts the state description length of all problems in the competition. Manually encoding some of the domains and comparing the results we often failed to devise a smaller state description length by hand.

Almost all of the execution time is spent on exploring predicate space. All the other phases added together never took more than a second of execution time. The time spent on exploring predicate space is not necessarily lost. When symbolically exploring planning space using BDDs, the operators need to be instantiated anyway for building the transition function, and if we keep track of all operator instantiations in the exploration phase this process can be sped up greatly.

## 6.9 Related Work and Conclusion

There is some work in literature dealing with reformulation of planning problems. However, research mainly concentrates on inferring state invariants instead of minimizing the state description length.

Fox and Long, for example, have contributed several suggestions that have been implemented in the planner *Stan* (for STate ANalysis) [243]. The project is based on Graphplan [39] and uses a variety of techniques to exhibit domain-dependent information. In this context the automatic inference of state invariants is important. The pre-processor *Tim* (Type Inference Module) explores planning domains in order to find typings of untyped parameters [129]. The information is found by an algorithm starting with a projection of actions to their parameters establishing so-called *properties*, i.e. predicates together with the argument position filled by the objects. Given the properties and operators, transition rules are inferred (e.g. $\texttt{on}_1 \rightarrow \texttt{on}_1$ in Logistics) which constitute a finites state machine corresponding to the property exchanges. Types are found by exploration of membership patterns starting with the initial set.

Given the inferred type specification, in an additional analysis step three major state invariants can be found: *identity invariants*, *membership invariants* and *unique state invariants*. E.g in *Blocks World* we have invariants of the form $\forall x, y, z\ \texttt{on}(y, x) \wedge \texttt{on}(z, x) \rightarrow y = z$, $\forall x\ \exists y\ \texttt{on}(y, x) \vee \texttt{clear}(x)$, and $\forall x\ \neg(\exists y\ \texttt{on}(y, x) \wedge \texttt{clear}(x))$ Furthermore, *Tim* infers *cardinality constraints* such as $|\{x | \texttt{at-robot}(x)\}| = 1$ in *Gripper*. Tim is sound but not complete, i.e., it will find correct invariants but not all of them. Very recent unpublished work by Fox and Long focus *Mobile Analysis*, which constructs maps of locations that can be navigated by a mobile through an operator schema that gives the mobility.

The problem of finding state invariants is also addressed by Gerevini and Schubert [141]. Their planner *Discoplan* discovers two kinds of invariance rules, *single-valued*

and *implicative constraints*. For example we have $\texttt{on}(x, y) \wedge y \neq z \Rightarrow \neg\texttt{on}(x, z)$ and $\texttt{on}(x, y) \wedge y \neq \texttt{table} \Rightarrow \neg\texttt{clear}(y)$ in *Blocks World*. While *Tim* improves explorations in *Graphplan*, in case of *Discoplan* the invariants improve satisfiability planning such as in *Satplan* [204]. Satplan itself is closely related to our approach of symbolically exploring planning space with BDDs, since both algorithms rely on a specification of the problem with boolean formulae.

The information gathered by *Tim* and *Discoplan* can be compared to our approach of balanced predicates and constraining the domains of predicates, since the presented algorithms exhibit domain-dependent knowledge leading to problem invariants as shown in the given example. As highlighted above the encoding in *Logistics* apparently give *identity invariants*. *membership invariants* and *unique state invariants* as well as some *single-valued* and *implicative constraints*. Even *cardinality constraints* can be extracted from the encodings.

We conjecture that it is possible to extract the same invariants as in Tim and Discoplan from our encodings and that the knowledge inferred by our algorithms is more detailed, but there is an extraction process required to obtain the invariants from the encodings and to prove the assertion. On the other hand we think that the binary encoding length is probably the best performance measure to compare the inferred knowledge of different precompilers.

Literature reveals that an information gathering phase prior to search takes time. Through the efficiency of our approaches the time spent on these efforts is by far shorter than the time needed for constructing the transition function and the symbolic search phase itself. Automatically inferring problem-dependent knowledge in planning problems is challenging but an inevitable necessity for current state space search engines. The paper contributes efficient new algorithms based on symbolical manipulation and search. The promising results of BDD-based exploration according to the achieved encodings are given in [113]. We conclude that our approach to automatically infer compressed state descriptions mainly tailored to symbolic exploration reflects current research and could have a strong impact on current planning systems.

| | Movie | | Gripper | | Logistics | | Mystery | | Mprime | | Grid | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| problem | bits | sec | bits | sec | bits | sec | bits | sec | bits | sec | bits | sec |
| 1-01 | 6 | <1 | 11 | <1 | 42 | <1 | 28 | <1 | 32 | <1 | | |
| 1-02 | 6 | <1 | 15 | <1 | 56 | <1 | 117 | <1 | 121 | <1 | | |
| 1-03 | 6 | <1 | 19 | <1 | 98 | <1 | 77 | <1 | 89 | <1 | | |
| 1-04 | 6 | <1 | 23 | <1 | 115 | <1 | 50 | <1 | 63 | <1 | | |
| 1-05 | 6 | <1 | 27 | <1 | 35 | <1 | 86 | <1 | 96 | <1 | | |
| 1-06 | 6 | <1 | 31 | <1 | 174 | <1 | 148 | <1 | 179 | <1 | | |
| 1-07 | 6 | <1 | 35 | <1 | 95 | <1 | 82 | <1 | 126 | 1 | | |
| 1-08 | 6 | <1 | 39 | <1 | 254 | 1 | 90 | <1 | 142 | <1 | | |
| 1-09 | 6 | <1 | 43 | <1 | 184 | <1 | 83 | <1 | 93 | <1 | | |
| 1-10 | 6 | <1 | 47 | <1 | 162 | <1 | 291 | <1 | 315 | <1 | | |
| 1-11 | 6 | <1 | 51 | <1 | 104 | 1 | 52 | 1 | 61 | 1 | | |
| 1-12 | 6 | <1 | 55 | <1 | 195 | <1 | 42 | <1 | 56 | <1 | | |
| 1-13 | 6 | <1 | 59 | <1 | 287 | 1 | 291 | <1 | 323 | 1 | | |
| 1-14 | 6 | <1 | 63 | <1 | 282 | 1 | 320 | 1 | 346 | 1 | | |
| 1-15 | 6 | <1 | 67 | <1 | 144 | <1 | 184 | <1 | 210 | 1 | | |
| 1-16 | 6 | <1 | 71 | <1 | 205 | <1 | 90 | <1 | 120 | <1 | | |
| 1-17 | 6 | <1 | 75 | <1 | 190 | <1 | 188 | <1 | 202 | <1 | | |
| 1-18 | 6 | <1 | 79 | <1 | 270 | 1 | 112 | 1 | 160 | 1 | | |
| 1-19 | 6 | <1 | 83 | <1 | 256 | <1 | 129 | <1 | 163 | <1 | | |
| 1-20 | 6 | <1 | 87 | <1 | 264 | 2 | 144 | <1 | 169 | 2 | | |
| 1-21 | 6 | <1 | | | 300 | 1 | 205 | <1 | 230 | <1 | | |
| 1-22 | 6 | <1 | | | 530 | 3 | 234 | 1 | 283 | 1 | | |
| 1-23 | 6 | <1 | | | 166 | <1 | 157 | <1 | 186 | <1 | | |
| 1-24 | 6 | <1 | | | 336 | <1 | 229 | <1 | 263 | 1 | | |
| 1-25 | 6 | <1 | | | 343 | 5 | 23 | <1 | 26 | <1 | | |
| 1-26 | 6 | <1 | | | 382 | 3 | 67 | <1 | 86 | <1 | | |
| 1-27 | 6 | <1 | | | 604 | 4 | 63 | <1 | 67 | <1 | | |
| 1-28 | 6 | <1 | | | 818 | <1 | 38 | <1 | 41 | <1 | | |
| 1-29 | 6 | <1 | | | 566 | 1 | 74 | <1 | 86 | <1 | | |
| 1-30 | 6 | <1 | | | 470 | 8 | 109 | <1 | 117 | 1 | | |
| 2-01 | | | | | 26 | <1 | | | 120 | <1 | 67 | <1 |
| 2-02 | | | | | 28 | <1 | | | 84 | <1 | 83 | <1 |
| 2-03 | | | | | 39 | <1 | | | 269 | <1 | 93 | <1 |
| 2-04 | | | | | 64 | <1 | | | 129 | <1 | 107 | <1 |
| 2-05 | | | | | 63 | <1 | | | 46 | <1 | 139 | 1 |

Table 6.2: Length of state encodings and elapsed time of the AIPS'98 benchmark set. The data was generated on a Sun Ultra Sparc Station.

# Paper 7

# The Model Checking Integrated Planning System

Stefan Edelkamp and Malte Helmert.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: {edelkamp,helmert}@informatik.uni-freiburg.de

*AI-Magazine*, pages 67–71, 2001.

**Abstract**

MIPS is a planning system that applies binary decision diagrams to compactly represent world states in a planning problem and efficiently explore the underlying state space. It was the first general planning system based on model checking methods. It can handle the STRIPS subset of the PDDL language and some additional features from ADL, namely negative preconditions and (universal) conditional effects. At the AIPS 2000 conference, MIPS was one of five planning systems to be awarded for "Distinguished Performance" in the fully automated track. This article gives a brief introduction to BDDs and explains the basic planning algorithm employed by MIPS, using a simple logistics problem as an example.

With the Model Checking Integrated Planning System MIPS, model checking has eventually approached classical AI planning. It was the first planning system based on formal verification techniques that turned out to be competitive with the various Graphplan- or SAT-based approaches on a broad spectrum of domains.

MIPS uses binary decision diagrams (BDDs, introduced by Bryant [50]) to compactly store and operate on sets of states. More precisely, it applies reduced ordered binary decision diagrams, which we will refer to simply as BDDs for the rest of this article.

Its main strength compared to other, similar approaches lies in its precompiling phase, which infers a concise state representation by exhibiting knowledge that is implicit in the description of the planning domain [101]. This representation is then used to carry out an accurate reachability analysis without necessarily encountering exponential explosion of the representation.

The original version of MIPS, presented at ECP99, was capable of handling the STRIPS subset of PDDL. It was later extended to handle some important features of ADL, namely domain constants, types, negative preconditions and universally quantified conditional effects.

Other extensions include two additional search engines based on heuristics, one incorporating a single-state hill-climbing technique very similar to Hoffmann's FF, the other one making use of BDD techniques, thus combining heuristic search with symbolic representations. However, as the former does not contribute many new ideas, its merits mainly lying in the combination of Hoffmann's heuristic estimate with the preprocessing techniques of MIPS, we won't dwell on it.

Neither will we say much about the symbolic heuristic search techniques included in MIPS, namely the BDDA* and Pure BDDA* algorithms, as those were disabled in the AIPS 2000 planning competition in favor of the original MIPS planning algorithm, partly because it turned out to perform better on some domains, partly because it always yields optimal (sequential) plans, which we consider an important property of the planner that counterbalances some of its weaknesses in performance compared to other current planning systems such as FF. Readers interested in those parts of the MIPS planning system are referred to Edelkamp and Helmert [102].

So in the following sections we will cover the core of MIPS, illustrating its basic techniques with a very simple example.

## 7.1 BDDs: Why and For What?

MIPS is based on satisfiability checking. This is indeed not a new idea. However, MIPS was the first SAT-based planning system to make use of binary decision diagrams to avoid (or at least lessen) the costs associated with the exponential blowup of the Boolean formulae involved as problem sizes get bigger. Since the early days of MIPS, other planning systems based on BDDs have emerged, most notably Fourman's PROPPLAN and Störr's BDDPLAN. We believe that the key advantage of MIPS compared to those systems lies in its preprocessing algorithms.

So it looks like BDDs are currently considered an interesting topic in AI Planning. Why is that? There is no doubt about the usefulness of this data structure. Nowadays, BDDs are a fundamental tool in various research areas, such as model checking and the synthesis and verification of hardware circuits. In AI Planning, they are mainly useful

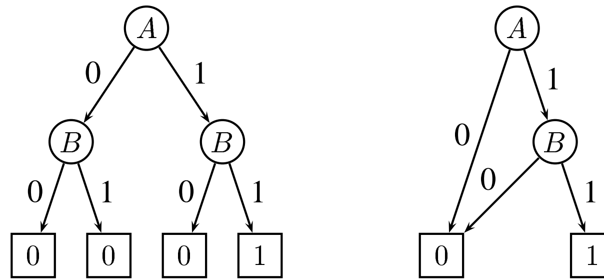


Figure 7.1: Two equivalent BDDs, a non-reduced and a reduced one. The "1" sink can only be reached by following the edges labeled "1" from $A$ and $B$, thus the represented boolean function $\psi(A, B)$ evaluates to true if and only if $A$ and $B$ are true.

because of their ability to efficiently represent huge sets of states commonly encountered in state-space search.

Without going into too much detail, a BDD is a data structure for efficiently representing Boolean functions, mapping bit strings of a fixed length to either "true" or "false". A BDD is a directed acyclic graph with a single root node and two sinks, labeled "1" and "0", respectively. For evaluating the represented function for a given input, a path is traced from the root node to one of the sinks, quite similar to the way decision trees are used. What distinguishes BDDs from decision trees is the use of certain reductions, detecting unnecessary variable tests and isomorphisms in subgraphs, leading to a unique representation that is polynomial in the length of the bit strings for many interesting functions. Figure 7.1 provides an example.

Among the operations supported by current BDD packages are all usual Boolean connectors such as "and" and "or", as well as constant time satisfiability and equality checking. MIPS uses the "Buddy" package by Jørn Lind-Nielsen, which we considered particularly useful for our purposes because of its ability to form groups of several Boolean variables to easily encode finite domain integers.

In MIPS, BDDs are used for two purposes: Representing sets of states and representing state transitions.

## 7.2 BDDs for Representing Sets of States

Given a fixed-length binary code for the state space of a planning problem, BDDs can be used to represent the characteristic function of a set of states (which evaluates to true for a given bit string, i.e. state, if and only if it is a member of that set). The characteristic function can be identified with the set itself.

Unfortunately, there are many different possibilities to come up with an encoding of states in a planning problem, and the more obvious ones seem to waste a lot of space which often leads to bad performance of BDD algorithms. It seems worthwhile to spend some effort on finding a *good* encoding, so this is where the preprocessing of MIPS enters the stage.

Let us consider a very simple example of a planning problem where a truck is supposed to deliver a package from Los Angeles to San Francisco. The initial situation in PDDL notation is given by (PACKAGE package), (TRUCK truck), (LOCATION los-

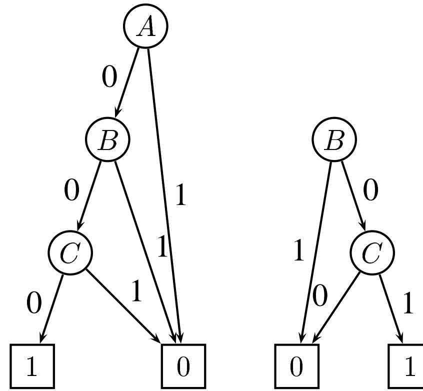


Figure 7.2: BDDs for the characteristic functions of the initial state, $init(A, B, C) = \neg A \wedge \neg B \wedge \neg C$, and of goal states, $goal(A, B, C) = \neg B \wedge C$.

angeles), (LOCATION san-francisco), (AT package los-angeles), and (AT truck los-angeles). Goal states have to satisfy the condition (AT package san-francisco). The domain provides three action schemata named LOAD to load a truck with a certain package at a certain location, the inverse operation UNLOAD, and DRIVE to move a truck from one location to another.

The first preprocessing step of MIPS will detect that only the AT (denoting the presence of a given truck or package at a certain location) and IN predicates (denoting that a package is loaded in a certain truck) are fluents and thus need to be encoded. The labeling predicates PACKAGE, TRUCK, LOCATION are not affected by any operator and thus do not need to be specified in a state encoding.

In a next step, some mutual exclusion constraints are discovered. In our case, we will detect that a given object will always be at or in at most one other object, so propositions such as (AT package los-angeles) and (IN package truck) are mutually exclusive.

This result is complemented by what we call fact space exploration: Ignoring negative (delete) effects of operators, we exhaustively enumerate all propositions that can be satisfied by any legal sequence of actions applied to the initial state, thus ruling out illegal propositions such as (IN los-angeles package), (AT package package) or (IN truck san-francisco).

Now all the information that is needed to devise an efficient state encoding schema for this particular problem is at the planner's hands. MIPS discovers that three Boolean variables $A$, $B$, and $C$ are needed. The first one is required for encoding the current city of the truck, where $A$ is set if (AT truck san-francisco) holds true, and $A$ is cleared otherwise, i.e. if (AT truck los-angeles) holds true. The other two variables $B$ and $C$ encode the status of the package: both are cleared if it is at Los Angeles, $C$ but not $B$ is set if it is at San Francisco, and $B$ but not $C$ is set if it is inside the truck.

We can now rephrase initial state and goal test as Boolean formulae, which can in turn be represented as BDDs: $\neg A \wedge \neg B \wedge \neg C$ denotes the initial situation, and the goal is reached in every state where $\neg B \wedge C$ holds true. The corresponding BDDs are illustrated in Figure 7.2.

## 7.3 BDDs for Representing State Transitions

What have we achieved so far? We were able to reformulate the initial and final situations as BDDs. As an end in itself, this does not help too much. We are interested in a sequence of actions (or *transitions*) that transforms an initial state into one that satisfies the goal condition.

Transitions are formalized as relations, i.e. as sets of tuples of predecessor and successor states, or alternatively as the characteristic function of such sets, Boolean formulae using variables $A$, $B$, $C$ for the old situation and $A'$, $B'$, $C'$ for the new situation. For example, the action (DRIVE truck los-angeles san-francisco), which is applicable if and only if the truck currently is in Los Angeles, and has as its effect a change of location of the truck, not altering the status of the package, can be formalized using the Boolean formula $\neg A \wedge A' \wedge (B \leftrightarrow B') \wedge (C \leftrightarrow C')$.

By conjoining this formula with any formula describing a set of states using variables $A$, $B$ and $C$ introduced before and querying the BDD engine for the possible instantiations of ($A'$, $B'$, $C'$), we can calculate all states that can be reached by driving the truck to San Francisco in some state from the input set. This, put shortly, is the relational product operator that is used at the core of MIPS to calculate a set of successor states from a set of predecessor states and a transition relation. Of course, we have more than one action at our disposal (otherwise planning would not be all that interesting), so rather than using the transition formula denoted above, we will build one such formula for each feasible action (adding a no-op action for technical reasons) and calculate the disjunction of those, illustrated in Figure 7.3.

Doing this in our example, starting from the set containing only the initial state, we get a set of three states (the initial state, one state where the truck has moved and one where the package was picked up), represented by a BDD with three internal nodes. Repeating this process, this time starting from the state set just calculated, we get a set of four states represented by a BDD with a single internal node, and a third iteration finally yields a state where the goal has been reached (Figure 7.4). This can be tested by building the conjunction of the current state set and goal state BDDs and testing for satisfiability.

By keeping the intermediary BDDs, a legal sequence of states linking the initial state to a goal state can then easily be extracted, which in turn can be used to find a corresponding sequence of actions.

## 7.4 Evaluation of the MIPS Algorithm

It is not hard to see that, given enough computational and memory resources, MIPS will find a correct plan if one exists. As it performs a breadth-first search in the state space, the first solution found will consist of a minimal number of steps. If no solution exists, this will also be detected - the breadth first search will eventually reach a fixpoint, which can easily be detected by comparing the successor BDD to the predecessor BDD after calculating the relational product. Thus, the algorithm is complete and optimal.

However, it is not blindingly fast, so various efforts were made to speed it up, mostly well-known standard techniques in symbolic search such as forward set simplification. A bigger gain in efficiency was achieved by using bidirectional search, which can be incorporated into the algorithm in a straight-forward fashion. One problem that arises in

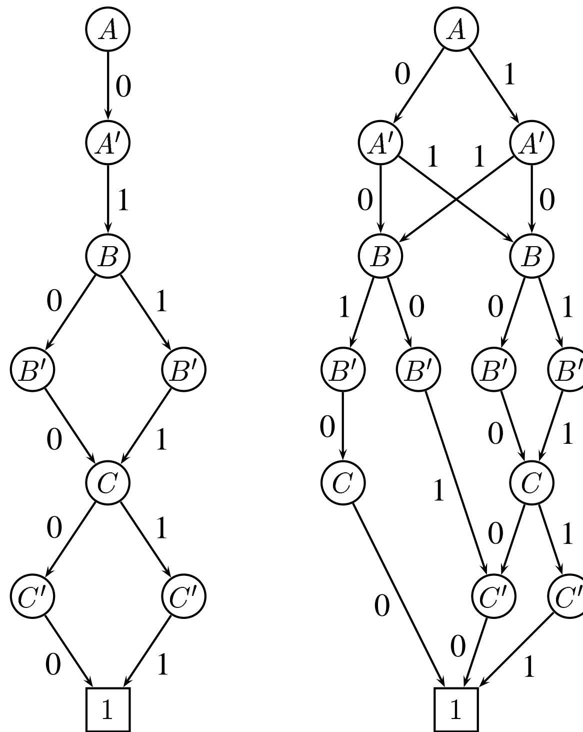


Figure 7.3: The left BDD represents the single action (DRIVE truck los-angeles san-francisco), the right one the disjunctions of all possible actions and thus the complete transition relation. The “0” sink and edges leading to it have been omitted for aesthetic reasons.

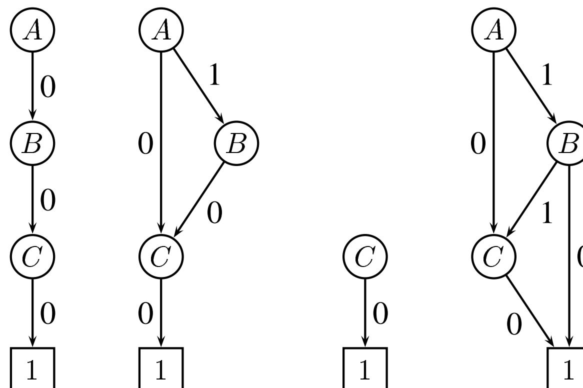


Figure 7.4: BDDs representing the set of reachable states after zero, one, two, and three iterations of the exploration algorithm. Note that the size (number of internal nodes) of a BDD does not necessarily grow with the number of states represented. Again, edges leading to the “0” sink have been omitted.

this context is that in some planning domains, backward iterations are far more expensive than forward iterations, and it is not trivial to decide when to perform which. We tried three different metrics to decide on the direction of the next exploration step: BDD size, number of states encoded, and time spent on the last exploration step in that direction. In our experiments, the last metric turned out to be most effective.

## 7.5 Outlook

As for the basic exploration algorithm, big improvements leading to a dramatically better performance are not to be expected for the near future, with the possible exception of transition function splitting, which still needs to be incorporated into the system.

From the algorithmic repertoire of MIPS, the heuristic symbolic search engine, which up to now has produced promising results but is still lacking in some domains, is getting most attention at the moment [41]. It might also be worthwhile to investigate the issue of optimal parallel plans, building on the work done by Haslum and Geffner for HSP [163].

Another research aim is the development of precomputed, informative and admissible estimates for explicit and symbolic search based on heuristic pattern databases.

The single most important area of interest, however, is certainly the extension of MIPS to more general flavours of planning such as conformant or strong cyclic planning where the strengths of symbolic methods are much more apparent than in the classical scenario [62, 63].

# Paper 8

# Directed Symbolic Exploration and its Application to AI-Planning

Stefan Edelkamp.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 84–92, 2001.

**Abstract**

In this paper we study traditional and enhanced BDD-based exploration procedures capable of handling large planning problems. On the one hand, reachability analysis and model checking have eventually approached AI-Planning. Unfortunately, they typically rely on uninformed *blind* search. On the other hand, heuristic search and especially lower bound techniques have matured in effectively directing the exploration even for large problem spaces. Therefore, with heuristic symbolic search we address the unexplored middle ground between single state and symbolic planning engines to establish algorithms that can gain from both sides. To this end we implement and evaluate heuristics found in state-of-the-art heuristic single-state search planners.

## 8.1 Introduction

One currently very successful trend in deterministic fully-automated planning is heuristic search. The search space incorporates states as lists of instantiated predicates (also called atoms or fluents). The success of the heuristic search correlates with the quality of the estimate; the more informed the heuristic the better the achieved results. Heuristic search planners have outperformed other approaches on a sizable collection of deterministic domains. In the fully automated track of the AIPS-2000 planning competition (`http://www.cs.toronto.edu/aips2000`) chaired by Fahim Baccus the System FF (by Hoffmann) was awarded for outstanding performance while HSP2 (by Geffner and Bonet), STAN (by Fox and Long), and MIPS (by Edelkamp and Helmert) were placed shared second.

Historically, the first heuristic search planner was HSP [42], which also competed in AIPS-1998. HSP computes the heuristic values of a state by adding (or maximizing) depth values for each fluent for an overestimating (or admissible) estimate. These values are retrieved from the fix point of a relaxed exploration. Since the technique is similar to the first phase of building the layered graph structure in GRAPHPLAN [39], HSPr [41] extends the approach by regression/backward search and excludes *mutuals* similar to the original planning graph algorithm. In the competition version HSP2 of the planner the *max-pair heuristic* computes a distance value to the goal for each pair of atoms. The underlying search algorithm is a weighted version of A* [287] implementing a higher influence of the heuristic by the cost of non-optimal solutions. Due to the observed overhead at run-time, high-order heuristics have not been applied yet.

HSP has inspired the planners GRT [302] and FF [177] and influenced the development of the planners STAN and MIPS. In AIPS-2000 the heuristic of GRT was too weak to compete with the improvements applied in HSP2 and in FF (for fast-forward planning). FF solves a relaxed planning problem for *every* encountered state in a combined forward *and* backward traversal. Therefore, the *FF-Heuristic* is an elaboration to the *HSP-Heuristic*, since the latter only considers the first phase. The efforts in computing a very accurate heuristic estimate correlates with data in solving Sokoban [199], which applies a $O(n^3)$ estimate, and suggests that even involved work for improving the heuristic pays off. With *enforced hill climbing* FF further employs another search strategy and reduces the explored portion of search space. It makes use of the fact that phenomena like big plateaus or local minima do not occur very often in benchmark planning problems. STAN is a hybrid of two strategies: The GRAPHPLAN-based algorithm and a forward planner using a heuristic function based on the length of the relaxed plan (as in HSP and FF). STAN performs a domain analysis techniques to select between these strategies. *Generic Types* automatically choose an appropriate algorithm for problem instance at hand [244].

An orthogonal approach in tackling huge search spaces is a symbolic representation of sets of states. The SATPLAN approach by Kautz and Selman [204] has shown that representional issues can be resolved by parsing the planning domain into a collection of Boolean formulae (one for each depth level). The system BLACKBOX, a hybrid planner based on merging SATPLAN with GRAPHPLAN, performed well on AIPS-1998, but failed to solve as many problems as the heuristic search planners on the domains in AIPS-2000. However, it should be denoted that the results of SATPLAN (GRAPHPLAN) are optimal in the number of sequential (parallel) steps, while heuristic search planners tend

to overestimate in order to cope with state space sizes of $10^{20}$ and beyond.

Although efficient satisfiability solvers have been developed in the last decade, the blow-up in the size of the formulae even for simple planning domains calls for a concise representation. This leads to reduced ordered binary decision diagrams (BDDs) [50], an efficient data structure for Boolean functions. Through their unique representation BDDs are effectively applied to the synthesis and verification of hardware circuits [49] and incorporated within the area of *model checking* [51]. Nowadays BDDs are a fundamental tool in various research areas of computer science and very recently BDDs are encountering AI-research topics like *heuristic search* [112] and *planning* [145]. The diverse research aspects of *traditional STRIPS planning* [113], *non-deterministic planning* [60], *universal planning* [63], and *conformant planning* [62] indicate the wide range of BDD-related planning.

The planner MIPS [102] uses BDDs to compactly store and maintain sets of propositionally represented states. The concise state representation is inferred in an analysis prior to the search and, by utilizing this representation, accurate reachability analysis and backward chaining are carried out without necessarily encountering exponential representation explosion. MIPS was originally designed to prove that BDD-based exploration methods are an efficient means for implementing a domain-independent planning system with some nice features, especially guaranteed optimality of the plans generated. If problems become harder and information on the solution length is available, MIPS invokes its incorporated heuristic single state search engine (similar to FF), thus featuring two entirely different planning algorithms, aimed to assist each other on the same state representation. The other two BDD planners in AIPS-2000, BDDPLAN [182] and PROPPLAN [128], lack the precompiling phase of MIPS. Therefore, these approaches were too slow for traditional STRIPS problems. Moreover, a single state extension to their planners has not being provided. In the generalized ADL settings, however, PROPPLAN has proven to be effective compared with the FF approach, which solves more problems in less time, but fails to find optimal solutions.

This paper extends the idea of BDD representations and exploration in the context of heuristic search. The heuristic estimate is based on subpositions (called patterns) calculated prior to the search. Therefore, the heuristic is a form of a pattern database with planning patterns corresponding to (one or a collection of) fluents. This heuristic will be integrated into a BDD-based version of the A* algorithm, called BDDA*. Moreover, we alter the concept of BDDA* to *pure heuristic search* which seems to be more suited at least to some planning problems. Thereby, we allow non-optimistic heuristics and sacrifice optimality but succeed in searching larger problem spaces. The paper is structured as follows. First of all, we give a simple planning example and briefly introduce BDDs basics. Thereafter, we turn to the exploration algorithms, starting with blind search then turning to the directed approach BDDA*, its adaption to planning, and its refinement for *pure heuristic search*. We end with some experimental data and draw conclusions.

## 8.2 BDD Representation

Let us consider an example of a planning problem. A truck has to deliver a package from Los Angeles to San Francisco. In STRIPS notation the start state is given by `(PACKAGE package)`, `(TRUCK truck)`,

(LOCATION los-angeles), (LOCATION san-francisco), (AT package los-angeles), and (AT truck los-angeles) while the goal is specified by (AT package san-francisco). We have three operator schemas in the domain, namely LOAD (for loading a truck with a certain package at a certain location), UNLOAD (the inverse operation), and DRIVE (a certain truck from one city to another). The operator schemas are expressed in form of preconditions and effects.

The precompiler to infer a small state encoding consists of three phases [101]. In a first *constant predicate* phase it observes that the predicates PACKAGE, TRUCK and LOCATION remain unchanged by the operators. In the next *merging* phase the precompiler determines that at and in should be encoded together, since a PACKAGE can exclusively be at a LOCATION or in a TRUCK. By *fact space exploration* (a simplified but complete exploration of the planning space) the following fluents are generated: (AT package los-angeles), (AT package san-francisco), (AT truck los-angeles), (AT truck san-francisco), and (IN package truck). This leads to a total encoding length of three bits. Using two bits $x_0$ and $x_1$ the fluents (AT package los-angeles), (AT package san-francisco), and (IN package truck) are encoded with 00, 01, and 10, respectively, while the variable $x_2$ represents the fluents (AT truck los-angeles) and (AT truck san-francisco).

Therefore, a Boolean representation of the start state is given by $\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_2}$ while the set of goal states is simply formalized with the expression $\overline{x_0} \wedge x_1$. More generally, for a set of states $S$ the *characteristic function* $\phi_S(a)$ evaluates to *true* if $a$ is the binary encoding of one state $x$ in $S$. As the formula for the set of goal states indicate, the symbolic representation for a large set of states is typically smaller than the cardinality of the represented set.

Since the satisfiability problem for Boolean formulae is NP hard, binary decision diagrams are used to for their efficient and unique graph representation. The nodes in the directed acyclic graph structure are labeled with the variables to be tested. Two outgoing edges labeled *true* and *false* direct the evaluation process with the result found at one of the two sinks. We assume a fixed variable ordering on every path from the root node to the sink and that each variable is tested at most once. The BDD size can be exponential in the number of variables but, fortunately, this effect rarely appears in practice. The satisfiability test is trivial and given two BDDs $G_f$ and $G_g$ and a Boolean operator $\otimes$, the BDD $G_{f \otimes g}$ can be computed efficiently. The most important operation for exploration is the *relational productrelational product* of a set of variables $v$ and two Boolean functions $f$ and $g$. It is defined as $\exists v \ (f \wedge g)$. Since existential quantification of one variable $x_i$ in a Boolean function $f$ is equal to disjunction $f_{\overline{x_i}} \vee f_{x_i}$, the quantification of $v$ results in a sequence of subproblem disjunctions. Although computing the relational product is NP-hard, specialized algorithms have been developed leading good results for many practical applications.

An operator can also be seen as an encoding of a set. The *transition relation* $T$ is defined as the disjunction of the characteristic functions of all pairs $(x', x)$ with $x'$ being the predecessor of $x$. For the example problem, (LOAD package truck los-angeles) corresponds to the pair $(00|0, 10|0)$ and (LOAD package truck san-francisco) to $(01|1, 10|1)$. Subsequently, the UNLOAD operator is given by $(10|0, 00|0)$ and $(10|1, 10|1)$. The DRIVE action for the truck is represented by the strings $(00|*, 00|*)$ $(01|*, 01|*)$, and $(10*, 10|*)$ with $* \in \{0, 1\}$. For a concise BDD representa-

tion (cf. Figure 8.1) the variable ordering is chosen that the set of variable in $x'$ and $x$ are *interleaved*, i.e. given in alternating order.

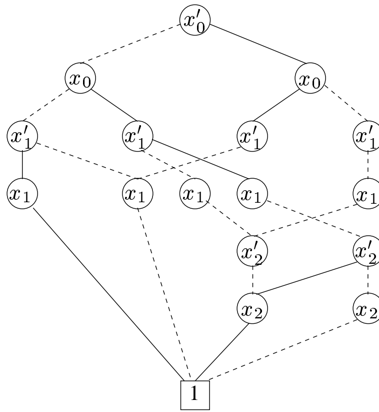

Figure 8.1: The transition relation for the example problem. For the sake of clarity, the *false* sink has been omitted. Dashed lines and solid lines indicate edges labeled *false* and *true*, respectively.

## 8.3 BDD-Based Blind Search

Let $S_i$ be the set of states reachable from the initial state $s$ in $i$ steps, initialized by $S_0 = \{s\}$. The following equation determines $\phi_{S_i}$ given both $\phi_{S_{i-1}}$ and the transition relation:

$$\phi_{S_i}(x) = \exists x' \, (\phi_{S_{i-1}}(x') \wedge T(x', x)).$$

The formula calculating the successor function is a relational product. A state $x$ belongs to $S_i$ if it has a predecessor $x'$ in the set $S_{i-1}$ and there exists an operator which transforms $x'$ into $x$. Note that on the right hand side of the equation $\phi$ depends on $x'$ compared to $x$ on the left hand side. Thus, it is necessary to substitute $x$ with $x'$ in $\phi_{S_i}$ beforehand, which can be achieved by a simple textual replacement of the node labels in the diagram structure. In order to terminate the search, we successively test, whether a state is represented in the intersection of the set $S_i$ and the set of goal states $G$ by testing the identity of $\phi_{S_i} \wedge \phi_G$ with the trivial zero function. Since we enumerated $S_0, \ldots, S_{i-1}$ the iteration index $i$ is known to be the optimal solution length.

Let *Open* be the representation of the search horizon and *Succ* the BDD for the set of successors. Then the algorithm can be realized as the pseudo-code Figure 8.2 suggests.

This simulates a breadth-first exploration and leads to three iterations for the example problem. We start with the initial state represented by a BDD of three inner nodes for the function $\overline{x_0} \wedge \overline{x_1} \wedge \overline{x_2}$. After the first iteration we get a BDD size of four representing three states and the function $(\overline{x_0} \wedge \overline{x_1}) \vee (x_0 \wedge \overline{x_1} \wedge \overline{x_2})$. The next iteration leads to four states in a BDD of one internal node for $\overline{x_1}$, while the last iteration results in a BDD containing a goal state.

**procedure** *Breadth-First Search*
  *Open* $\leftarrow \phi_{\{s\}}$
  **do**
    *Succ* $\leftarrow \exists x' \ (Open(x') \wedge T(x', x))$
    *Open* $\leftarrow$ *Succ*
  **while** (*Open* $\wedge \ \phi_G \equiv 0$)

Figure 8.2: Breadth-first search implemented with BDDs.

### 8.3.1 Bidirectional Search

In backward search we start with the goal set $B_0$ and iterate until we encounter the start state. We take advantage of the fact that $T$ has been defined as a relation. Therefore, we iterate according to the formula $\phi_{B_i}(x') = \exists x \ (\phi_{B_{i-1}}(x) \wedge T(x', x))$. In bidirectional breadth-first search forward and backward search are carried out concurrently. On the one hand we have the forward search frontier $F_f$ with $F_0 = \{s\}$ and on the other hand the backward search frontier $B_b$ with $B_0 = G$. When the two search frontiers meet ($\phi_{F_f} \wedge \phi_{B_b} \not\equiv 0$) we have found an optimal solution of length $f + b$. With the two horizons *fOpen* and *bOpen* the algorithm can be implemented as shown in Figure 8.3.

**procedure** *Bidirectional Breadth-First Search*
  *fOpen* $\leftarrow \phi_{\{s\}}$; *bOpen* $\leftarrow \phi_G$
  **do**
    **if** (*forward*)
      *Succ* $\leftarrow \exists x' \ (fOpen(x') \wedge T(x', x))$
      *fOpen* $\leftarrow$ *Succ*
    **else**
      *Succ* $\leftarrow \exists x \ (bOpen(x) \wedge T(x', x))$
      *bOpen* $\leftarrow$ *Succ*
  **while** (*fOpen* $\wedge$ *bOpen* $\equiv 0$)

Figure 8.3: Bidirectional BFS implemented with BDDs.

### 8.3.2 Forward Set Simplification

The introduction of a list *Closed* containing all states ever expanded is an apparent very common approach in single state exploration to avoid duplicates in the search. The memory structure is realized as a transposition table. For symbolic search this technique is called *forward set simplification* (cf. Figure 8.4).

The effect in the given example is that after the first iteration the number of states shrinks from three to two while the new BDD for $(\overline{x_0} \wedge \ \overline{x_1} \wedge \ x_2) \ \vee \ (x_0 \wedge \ \overline{x_1} \wedge \ \overline{x_2})$ has five inner nodes. For the second iteration only one newly encountered state is left with three inner BDD nodes representing $x_0 \wedge \ \overline{x_1} \wedge \ \overline{x_2}$. Forward set simplification

**procedure** *Forward Set Simplification*
*Closed* $\leftarrow$ *Open* $\leftarrow \phi_{\{s\}}$
**do**
*Succ* $\leftarrow \exists x' \ (Open(x') \wedge T(x', x))$
*Open* $\leftarrow$ *Succ* $\wedge \neg$ *Closed*
*Closed* $\leftarrow$ *Closed* $\vee$ *Succ*
**while** (*Open* $\wedge \phi_G \equiv 0$)

Figure 8.4: Symbolic BFS with *forward set simplification*.

terminates the search in case of a complete planning space exploration. Note that any set in between the successor set *Succ* and the simplified successor set *Succ* $-$ *Closed* will be a valid choice for the horizon *Open* in the next iteration. Therefore, one may choose a set $R$ that minimizes the BDD representation instead of minimizing the set of represented states. Without going into details we denote that such image size optimizing operators are available in several BDD packages [71].

## 8.4 BDD-Based Directed Search

Before turning to the BDD-based algorithms for directed search we take a brief look at Dijkstra's single-source shortest path algorithm, *Dijkstra* for short, which finds a solution path with minimal length within a weighted problem graph [80]. *Dijkstra* differs from breadth-first search in ranking the states next to be expanded. A priority queue is used, in which the states are ordered with respect to an increasing $f$-value. Initially, the queue contains only the initial state $s$. In each step the state with the minimum merit $f$ is dequeued and expanded. Then the successor states are inserted into the queue according to their newly determined $f$-value. The algorithm terminates when the dequeued element is a goal state and returns the minimal solution path.

As said, BDDs allow sets of states to be represented very efficiently. Therefore, the priority queue *Open* can be represented by a BDD based on tuples of the form (*value*, *state*). The variables should be ordered in a way which allows the most significant variables to be tested at the top. The variables for the encoding of *value* should have smaller indices than the variables encoding *state*, since this leads to small BDDs and allows an intuitive understanding of the BDD and its association with the priority queue.

Let the *weighted transition relation* $T(w, x', x)$ evaluates to 1 if and only if the step from $x'$ to $x$ has costs $w$ (encoded in binary). The symbolic version of Dijkstra (cf. Figure 8.5) now reads as follows. The BDD *Open* is set to the representation of the start state with value zero. Until we find a goal state in each iteration we extract *all* states with minimal $f$-value $f_{\min}$, determine the successor set and update the priority queue. Successively, we compute the minimal $f$-value $f_{\min}$, the BDD *Min* of all states in the priority queue with value $f_{\min}$, and the BDD of the remaining set of states. If no goal state is found, the variables in *Min* are substituted as above before the (weighted) transition relation $T(w, x', x)$ is applied to determine the BDD for the set of successor states. To attach new $f$-values to this set we have to retain the old $f$-value $f_{\min}$ and in

**procedure** Symbolic-Version-of-Dijkstra
  *Open*$(f, x) \leftarrow (f = 0) \wedge \phi_{S^0}(x)$
  **do**
    $f_{\min} = \min\{f \mid f \wedge \mathit{Open} \neq \emptyset\}$
    *Min*$(x) \leftarrow \exists f\ (\mathit{Open} \wedge f = f_{\min})$
    *Rest*$(f, x) \leftarrow \mathit{Open} \wedge \neg\, \mathit{Min}$
    *Succ*$(f, x) \leftarrow \exists x', w\ (\mathit{Min}(x') \wedge$
      $T(w, x', x) \wedge \mathit{add}(f_{\min}, w, f))$
    *Open* $\leftarrow$ *Rest* $\vee$ *Succ*
  **while** $(\mathit{Open} \wedge \phi_G \equiv 0)$

Figure 8.5: Dijkstra's single-source shortest-path algorithm implemented with BDDs.

order to calculate $f = f_{\min} + w$. Finally, the BDD *Open* for the next iteration is obtained by the disjunction of the successor set with the remaining queue.

It remains to show how to perform the arithmetics using BDDs. Since the $f$-values are restricted to a finite domain, the Boolean function *add* with parameters $a$, $b$ and $c$ can be built being *true* if $c$ is equal to the sum of $a$ and $b$. A recursive calculation of *add*$(a, b, c)$ should be prefered:

$$\mathit{add}(a, b, c) = ((b = 0) \wedge\ (a = c)) \vee$$

$$\exists\, b', c'\ (\mathit{inc}(b', b)\ \wedge\ \mathit{inc}(c', c)\ \wedge\ \mathit{add}(a, b', c')),$$

with *inc* representing all pairs of the form $(i, i + 1)$. Therefore, symbolic breadth-first search can be applied to determine the fixpoint of *add*.

### 8.4.1 Heuristic Pattern Databases

For symbolically constructing the heuristic function a simplification $T'$ to the transition relation $T$ that regains tractability of the state space is desirable. However, obvious simplification rules might not be available. Therefore, in heuristic search we often consider relaxations of the problem that result in subpositions. More formally, a state $v$ is a *subposition* of another state $u$ if and only if the characteristic function of $u$ logically implies the characteristic function of $v$, e.g., $\phi_{\{u\}} = \overline{x_1} \wedge x_2 \wedge x_3 \wedge \overline{x_4} \wedge x_5$ and $\phi_{\{v\}} = x_2 \wedge x_3$ results in $\phi_{\{u\}} \Rightarrow \phi_{\{v\}}$. As a simple example take the Manhattan distance in sliding tile solitaire games like the famous Fifteen-Puzzle. It is the sum of solutions of single tile problems that occur in the overall puzzle.

More generally, a *heuristic pattern database* is a collection of pairs of the form (*value*, *pattern*) found by optimally solving problem relaxations that respect the subposition property [75]. The solution lengths of the patterns are then combined to an overall heuristic by taking the maximum (leading to an admissible heuristic) or the sum of the individual values (in which case we overestimate).

Heuristic pattern databases have been effectively applied in the domains of Sokoban [199], to the Fifteen-Puzzle [75], and to Rubik's Cube [215]. In single-state search heuristic pattern databases are implemented by hash table, but in symbolic search we have to

construct the estimator symollically, only using logical combinators and Boolean quantification.

Since heuristic search itself can be considered as the matter of introducing lower bound relaxations into the search process, in the following we will maximize the relaxed solution path values. The maximizing relation $max(a, b, c)$, evaluates to 1 if $c$ is the maximum of $a$ and $b$ and is based on the relation *greater*, since

$$max(a, b, c) = (greater(a, b) \wedge \ (a = c)) \vee$$

$$(\neg greater(a, b) \wedge \ (b = c)).$$

The relation $greater(a, b)$ itself might be implemeted by existential quantifying the add relation:

$$greater(a, b) = \exists t \ add(b, t, a)$$

Next we will find a way to automatically infer the heuristic estimate. To combine $n$ fluent pattern $p_1, \ldots, p_n$ with estimated distances $d_1, \ldots, d_n$ to the goal we use $n + 1$ additional slack variables $t_0, \ldots, t_n$ which are existenially quantified later on. We define subfunctions $H_i$ of the form

$$H_i(t_i, t_{i+1}, state) = (\neg \ p_i \ \wedge \ (t_i = t_{i+1}))\vee$$

$$(p_i \ \wedge \ max(d_i, t_i, t_{i+1})),$$

with $H_i(t_i, t_{i+1}, state)$ denoting the following relation: If the accumulated heuristic value up to fluent $i$ is $t_i$, then the accumulated value including fluent $i$ is $t_{i+1}$. Therefore, we can combine the subfunctions to the overall heuristic estimate as follows.

$$H(value{,}state) = \exists \ t_1, \ldots, t_n$$

$$(t_0 = 0) \wedge \ H(t_n, value{,}state) \ \wedge \ \bigwedge_{i=0}^{n-1} H_i(t_i, t_{i+1}, state).$$

In some problem graphs subpositions or patterns might constitute a feature in which every position containing it is unsolvable. These *dead-ends* are frequent in directed search problems like Sokoban and can be learned domain or problem specifically. Dead ends are heuristic patterns with an infinite heuristic estimate. Therefore, a dead end table $DT$ is the disjunction of the characteristic functions according to subpositions that are unsolvable. The integration of *dead-end tables* in the search algorithm is quite simple. For the BDD for $DT$ we assign the new horizon *Open* as

$$Open \wedge \neg(Open \Rightarrow DT) = Open \wedge \neg DT.$$

The simplest patterns in planning are fluents. The estimated distance of each single fluent $p$ to the goal is a heuristic value associated with $p$. We examine two heuristics.

**HSP-Heuristic:**

In HSP the values are recursively calculated by the formula $h(p) = min\{h(p), 1 + h(C)\}$ where $h(C)$ is the cost of achieving the conjunct $C$, which in case of HSPr is the list of preconditions. For determining the heuristic the planning space has been simplified by

omitting the delete effects. The algorithms in HSP and HSPr are variants of pure heuristic search incorporated with restarts, plateau moves, and overestimation.

The exploration phase to minimize the state description length in our planner has been extended to output an estimate $h(p)$ for each fluent $p$. Since we avoid duplicate fluents in the breadth-first *fact-space exploration*, with each encountered fluent we associate a depth by adding the value 1 to its predecessor. The quality of the achieved distance values are not as good as in HSPr since we are not concerned about mutual exclusions in any form. Giving the list of value/fluent pairs a symbolic representation of the sub-relations and the overall heuristic is computed.

In the example `(AT ball los-angeles)` and `(AT truck los-angeles)` have distance 0 from the initial state `(AT truck san-francisco)` `(IN ball truck)` have a depth of one and `(AT ball san-francisco)` has depth two. Figure 8.6 depicts the BDD representation of the overall heuristic function.

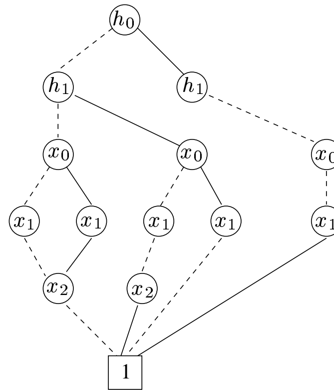


Figure 8.6: The BDD representation for the heuristic function in the example problem. In this case the individual pattern values have been maximized.

**FF-Heuristic:**

FF solves the relaxed planning problem (delete-facts omitted) with GRAPHPLAN online for each state. For estimation FF builds the plan graph *and* extracts a simplified solution by counting the number of instantiated operators that at least have to fire Since the branching factor is large (one state has up to hundreds of successors) by determining *helpful actions*, only a *relevant* part of all successors is considered. The overall search phase is entitled *enforced hill climbing*. Until the next smaller heuristic value is found a breadth-first search is invoked. Then the search process iterates with one state evaluating to this value.

In our planner we have (re-)implemented the FF-approach both to have an efficient heuristic single-state search engine at hand and to build an improved estimate for symbolic search. Since the FF approach is based on states and not on fluents, we cannot directly infer a symbolic version of the heuristic. We have to weaken the state-dependent character of the heuristic down to fluents. Moreover, simplifying the start state to a fluent may give no heuristic value at all, since the goal will not necessarily be reached by the relaxed exploration. Therefore, the estimate for each fluent is calculated by partitioning the goal state instead. Since we get improved distance estimates with respect to the initial state, we

obtain a heuristic for backward search. However this is no limitation, since the concept of STRIPS operators can be inverted, yielding a heuristic in the usual direction.

### 8.4.2 BDDA*

In *informed search* with every state in the search space we associate a lower bound estimate $h$. By reweighting the edges the algorithm of Dijkstra can be transformed into A*. The new weight $\hat{w}$ is set to the old one $w$ minus the $h$-value of the source node $x'$, plus the value of the target node $x$ resulting in the equation $\hat{w}(x', x) = w(x', x) - h(x') + h(x)$. The length of the shortest paths will be preserved and no new negative weighted cycle is introduced [70]. More formally, if we denote $\delta(s, g)$ for the length of the shortest path from $s$ to a goal state $g$ in the original graph, and $\hat{\delta}(s, g)$ the shortest path in the reweigthed graph then $w(p) = \delta(s, g)$ if and only if $\hat{w}(p) = \hat{\delta}(s, g)$.

The rank of a node is the combined value $f = g + h$ of the generating path length $g$ and the estimate $h$. The information $h$ allows us to search in the direction of the goal and its quality mainly influences the number of nodes to be expanded until the goal is reached.

In the symbolic version of A*, called BDDA*, the relational product algorithm determines all successor states in one evaluation step. It remains to determine their values. For the dequeued state $x'$ we have $f(x') = g(x') + h(x')$. Since we can access $f$, but usually not $g$, the new value $f(x)$ of a successor $x$ has to be calculated in the following way

$$f(x) = g(x) + h(x) = g(x') + w(x', x) + h(x) =$$

$$f(x') + w(x', x) - h(x') + h(x).$$

The estimator $H$ can be seen as a relation of tuples (*value*, *state*) which is *true* if and only if $h$(*state*)=*value*. We assume that $H$ can be represented as a BDD for the entire problem space. The cost values of the successor set are calculated according to the equation mentioned above. The arithmetics for *formula*$(h', h, w, f', f)$ based on the old and new heuristic value ($h'$ and $h$, respectively), and the old and new merit ($f'$ and $f$, respectively) are given as follows.

$$\textit{formula}(h', h, w, f', f) = \exists\, t_1, t_2\; add(t_1, h', f') \wedge$$

$$add(t_1, w, t_2)\ \wedge\ add(h, t_2, f).$$

The implementation of BDDA* is depicted in Figure 8.7. Since all successor states are reinserted in the queue we expand the search tree in best-first manner. Optimality and completeness is inherited by the fact that given an optimistic heuristic A* will find an optimal solution.

Given a uniform weighted problem graph and a consistent heuristic the worst-case number of iterations in BDDA* is $O(f^{*2})$, with $f^*$ being the optimal solution length [112]. In (a moderately difficult instance to) the Fifteen-Puzzle, the $4 \times 4$ version of the well-known sliding-tile $(n^2 - 1)$-Puzzles, a minimal solution was found by BDDA* within 176 iterations. With a breadth-first search approach it was impossible to find any solutions because of memory limitations. Already after 19 iterations more than 1 million BDD-nodes were needed to represent more than 1.4 million states.

To find the minimal solution in the first problem to Sokoban (6 balls) the BDDA* algorithm was invoked with a very poor heuristic, counting the number of balls not on a

**procedure** BDDA*
*Open*$(f, x) \leftarrow H(f, x) \wedge \phi_{S^0}(x)$
**do**
$f_{\min} = \min\{f \mid f \wedge Open \neq \emptyset\}$
*Min*$(x) \leftarrow \exists f \ (Open \wedge f = f_{\min})$
*Rest*$(f, x) \leftarrow Open \wedge \neg Min$
*Succ*$(f, x) \leftarrow \exists w, x' \ (Min(x') \wedge T(w, x', x) \wedge$
$\exists h' \ (H(h', x') \wedge \exists h \ (H(h, x) \wedge$
$formula(h', h, w, f_{\min}, f))))$
*Open* $\leftarrow$ *Rest* $\vee$ *Succ*
**while** $(Open \wedge \phi_G \equiv 0)$

Figure 8.7: A* implemented with BDDs.

goal position. Breadth-first search finds the optimal solution with a peak BDD of 250,000 nodes representing 61,000,000 states in the optimal number of 230 iterations. BDDA* with the heuristic leads to 419 iterations and to a peak BDD of 68,000 nodes representing 4,300,00 states. Note that even with such a poor heuristic, the number of nodes expanded by BDDA* is significantly smaller than in a breadth-first-search approach and their representation is more memory efficient. The number of represented states is up to 250 times larger than the number of BDD nodes.

### 8.4.3 Best-First-Search

A variant of BDDA*, called *Symbolic Best-First-Search*, can be obtained by ordering the priority queue only according to the $h$ values. In this case the calculation of the successor relation simplifies to $\exists x' \ (Min(x') \wedge T(x', x) \wedge H(f, x))$ as shown in Figure 8.8. The old $f$-value are replaced.

**procedure** Best-First-Search
*Open*$\leftarrow H(f, x) \wedge \phi_{S^0}$
**do**
$f_{\min} = \min\{f \mid f \wedge Open \neq \emptyset\}$
*Min*$(x) \leftarrow \exists f \ Open \wedge f = f_{\min}$
*Rest*$(f, x) \leftarrow Open \wedge \neg Min$
*Succ* $\leftarrow \exists x' \ (Min(x') \wedge T(x', x) \wedge H(f, x))$
*Open* $\leftarrow$ *Rest* $\vee$ *Succ*
**while** $(Open \wedge \phi_G \equiv 0)$

Figure 8.8: Best-first search implemented with BDDs.

Unfortunately, even for an optimistic heuristic the algorithm is not admissible and, therefore, will not necessarily find an optimal solution. The hope is that in huge problem

spaces the estimate is good enough to lead the solver into a promising goal direction. Therefore, especially heuristics with overestimizations can support this aim.

On solution paths the heuristic values eventually decrease. Hence, Best-first search profits from the fact that the most promising states are in the front of the priority queue, have a smaller BDD representation, and are explored first. This compares to BDDA* in which the combined merit on the solution paths eventually increases. A good trade-off between exploitation and exploration has to be found. In FF breadth-first search for the next heuristic estimate consolidates pure heuristic search for a complete search strategy.

Figure 8.9 depicts the different dequeued BDDs *Min* together with the encoded heuristic in the exploration phase of *Pure* BDDA* for the example problem.

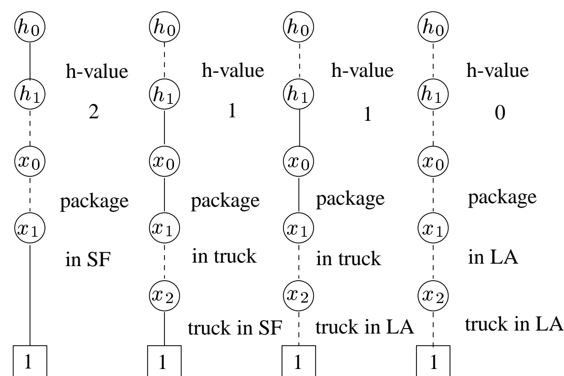


Figure 8.9: Backward exploration of the example problem in *Pure* BDDA*. In each iteration step the BDD *Min* with associated $h$-value is shown. Note that when using forward set simplification these BDDs additionally correspond to a snapshot of the priority queue *Open*.

## 8.5 Experiments

From given results on the different heuristic search planners [177] it can be obtained that heuristics pay off best in the *Gripper* and the *Logistics* domain.

### 8.5.1 Gripper

The effect of forward set simplification and optimization can best be studied in the scalable *Gripper* domain depicted in Table 8.1[1] When the problem instances get larger the additional computations pay off. In *Gripper* bidirectional search leads to no advantage since due to the symmetry of the problem the climax of the BDD sizes is achieved in the middle of the exploration. This is an important advantage to BDD-based exploration: Although the number of states grows continuously, the BDD representation might settle and become smaller. The data further suggests that optimizing the BDD structure with the proposed optimization is helpful only in large problems. *Gripper* is not a problem to *BDD*-based search, whereas it is suprisingly hard for GRAPHPLAN.

[1]The CPU-times in the experiments are given in seconds on a Linux-PC (Pentium III/450 MHz/128 MByte).

| | Solution Length | BFS | +B | +BF | +BFO |
|---|---|---|---|---|---|
| 1-1 | 11 | 0.00 | 0.01 | 0.01 | 0.01 |
| 1-2 | 17 | 0.01 | 0.01 | 0.02 | 0.02 |
| 1-3 | 23 | 0.02 | 0.03 | 0.02 | 0.02 |
| 1-4 | 29 | 0.03 | 0.03 | 0.04 | 0.04 |
| 1-5 | 35 | 0.04 | 0.04 | 0.07 | 0.07 |
| 1-6 | 41 | 0.06 | 0.06 | 0.08 | 0.08 |
| 1-7 | 47 | 0.08 | 0.08 | 0.11 | 0.14 |
| 1-8 | 53 | 0.12 | 0.13 | 0.19 | 0.20 |
| 1-9 | 59 | 0.35 | 0.36 | 1.33 | 1.58 |
| 1-10 | 65 | 0.72 | 1.93 | 2.06 | 2.15 |
| 1-11 | 71 | 1.27 | 2.33 | 2.36 | 2.43 |
| 1-12 | 77 | 1.95 | 3.21 | 3.05 | 3.13 |
| 1-13 | 83 | 2.80 | 3.91 | 3.48 | 3.49 |
| 1-14 | 89 | 3.80 | 5.04 | 4.28 | 4.36 |
| 1-15 | 95 | 4.93 | 6.26 | 5.29 | 5.43 |
| 1-16 | 101 | 6.32 | 7.21 | 6.41 | 6.07 |
| 1-17 | 107 | 7.72 | 8.94 | 7.26 | 7.52 |
| 1-18 | 113 | 9.82 | 10.91 | 8.65 | 8.61 |
| 1-19 | 119 | 24.73 | 26.11 | 15.28 | 15.35 |
| 1-20 | 125 | 34.59 | 36.73 | 20.41 | 20.08 |

Table 8.1: Solution lengths and computation times in solving Gripper with breadth-first bidirectional search, forward set simplification and optimization; $B$ abbreviates *bidirectional search*, $O$ denotes BDD image *optimization*, and $F$ is *forward set simplification*.

### 8.5.2 Logistics

Due to the first round results in AIPS-2000 it can be deduced that FF's, STAN's and MIPS's heuristic single search engine are state-of-the-art in this domain, but Logistics problems turn out to be suprisingly hard for BDD exploration and therefore a good benchmark domain for BDD inventions. For example Jensen's BDD-based planning system, called UMOP, fails to solve any of the AIPS-1998 (first-round) problems [196] and breadth-first search in *MIPS* yields only two domains to be solved optimally. This is due to high parallelism in the plans, since optimal parallel (Graphplan-based) planners, like IPP (by Köhler), Blackbox (by Kautz and Selman), Graphplan (by Blum and Furst), STAN (by Fox and Long) perform well on Logistics. Note, that heuristic search planners, such as (parallel) HSP2 with an IDA* like search engine loose their perfomance gains when optimality has to be preserved.

With best-first-search and the FF-Heuristic, however, we can solve 11 of the 30 problem instances. The dauting problem is that – due to the large minimized encoding size of the problems – the transition function becomes too large to be build. Therefore, the Logistics benchmark suite in the *Blackbox* distribution and in AIPS-2000 scale better. In AIPS-2000 we can solve the entire first set of problems with heuristic sybolic search and Table 8.2 visulizes the effect of best-first search for the *Logistics* suite of the *Blackbox*

distribution, in which all 30 problems have encodings of less than 100 bits. We measured the time, and the length of the found solution. $H_{add}^{HSP}$ and $H_{max}^{HSP}$ abbreviate best-first search according to the *add* and the *max* relation in the HSP-heuristic, respectively. $H_{add}^{FF}$ and $H_{max}^{FF}$ are defined analogously. The depicted times are not containing the efforts for determining the heuristic functions, which takes about a few seconds for each problem. Obviously, searching with the *max*-Heuristic achieves a better solution quality, but on the other hand it takes by far more time. The data indicates that on average the *FF-Heuristic* leads to shorter solutions and to smaller execution times. This was expected, since the average heuristic value per fluent in $H^{FF}$ is larger than in $H^{HSP}$, e.g. in the first problem it increases from 2.96 to 4.43 and on the whole set we measured an average increase of 41.25 % for the estimate.

The backward search component - here applied in the regression space (thus corresponding to forward search in progression space) is used as a breadth-first *target enlargement*. With higher search tree depths this approach definitely profits from the symbolic representation of states.

In best-first-search forward simplification is used to avoid recurrences in the set of expanded states. However, if the set of reachable states from the first bucket in the priority queue returns with failure, we are not done, since the set of goal states according to the minimum may not be reachable.

### 8.5.3 Planning as Model Checking

The model checking problem determines whether a formula is true in a concrete model and is based on the following issues [145]:

1. A domain of interest (e.g, a computer program or a reactive system) is described by a formal model.

2. A desired property of finite domain (e.g. a specification of a program, a safety requirement for a reactive system) is described by a formula typically using temporal logic.

3. The fact that a domain satisfies a desired property (e. g. that a reactive system never ends up in a dangerous state) is determined by checking whether or not the formula is true in the initial state of the model.

The crucial observation is that exploring (deterministic or non-deterministic) planning problem spaces is in fact a model checking problem. In model checking the assumed structure is described as a Kripke structure $(W, W_0, T, L)$, where $W$ is the set of states, $W_0$ the set of initial states, $T$ the transition relation and $L$ a labeling function that assigns to each state the set of atomic propositions which evaluate to *true* in this state. The properties are usually stated in a temporal formalism like linear time logic LTL (used in SPIN) or branching time logic CTL eventually enhanced with fairness constraints. In practice, however, the characteristics people mainly try to verify are simple safety properties expressible in all of the logics mentioned above. They can be checked through a simple calculation of all reachable states. An iterative calculation of Boolean expressions has to be performed to verify the formula *EF Goal* in the temporal logic CTL which is dual to the verification of *AG* $\neg$*Goal*. The computation of a (minimal) witness delivers a

solution. BDD-based planning approaches are capable of dealing with non-deterministic domains [63]. Due to the non-determinism the authors refer to plans as complete state action tables. Therefore, actions are included in the transition relation, resulting in a representation of the form $T(\alpha, x', x)$. The concept of *strong cyclic plans* expresses that from each state on a path a goal state is eventually reachable.

## 8.6 Conclusion and Outlook

Symbolic breadth-first search and BDDA* have been applied to the areas *search* [112] and *model checking* [303]. The experiments in *(heuristic) search* indicate the potential power of the symbolic exploration technique (in Sokoban) and the lower bound information (in the Fifteen Puzzle). In *model checking* we encounter a real-world problem of finding errors in hardware devices. BDD sizes of 25 million nodes reveal that even with symbolic representations we operate at the limit of main memory. However, this study of domain independent *planning* proves the generality of BDDA*.

The BDD representation of the space allows to reduce the planning problem to model checking: reachability analysis verifies the formula **EF** *Goal* in the temporal logic CTL.

The directed BDD-based search techniques bridge the gap between heuristic search planners and symbolic methods. Especially, best-first search and the FF-heuristic seem very promising to be studied in more detail and to be evaluated in other application areas. Due to off-line computation and restriction to one atom in the planning patterns, the symbolic HSP- and FF-heuristics are not as informative as their respective single-state correspondants in FF and HSP2, but, nevertheless, lead to good results.

The extension of the approach to planning patterns with more than one encoded atom is challenging. One possibility is a regressive breadth-first exploration through an abstraction of the state-space to build a pattern-estimate data-base. In [95] we show how this approach leads to a very good admissible estimate in explicite search. We have experimented with an extension to the *max-pair heuristic* with a weighted bipartite minimum-matching, but explicite pattern databases lead to better results. Together with the wide range of applicability, we expect that with the same heuristic information a symbolic planner is competitive with a explicite one if at least moderate-sized sets of states have to be explored. In future we will also consider other symbolic heuristic search algorithms such as *Symbolic Hill-Climbing* and *Symbolic Weighted A**.

| | BFS | | $H_{add}^{HSP}$ | | $H_{max}^{HSP}$ | | $H_{add}^{FF}$ | | $H_{max}^{FF}$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 25 | 0.66 | 30 | 0.06 | 25 | 1.05 | 30 | 0.92 | 25 | 0.49 |
| 2 | 24 | 121 | 27 | 5.33 | 24 | 129 | 31 | 1.27 | 26 | 3.52 |
| 3 | - | - | 29 | 3.30 | 26 | 35.98 | 28 | 1.18 | 26 | 30.22 |
| 4 | - | - | 59 | 6.53 | 52 | 37.10 | 59 | 3.49 | 52 | 22.74 |
| 5 | - | - | 52 | 5.64 | 42 | 4.56 | 51 | 3.11 | 43 | 3.41 |
| 6 | 42 | 72 | 63 | 7.22 | 51 | 67.18 | 64 | 2.45 | 52 | 11.37 |
| 7 | - | - | 83 | 14.89 | - | - | 80 | 11.87 | - | - |
| 8 | - | - | 84 | 19.14 | - | - | 80 | 15.05 | - | - |
| 9 | - | - | 84 | 13.07 | - | - | 80 | 8.94 | - | - |
| 10 | - | - | 47 | 13.93 | 40 | 484 | 45 | 8.15 | 40 | 421 |
| 11 | - | - | 54 | 10.10 | - | - | 52 | 7.30 | - | - |
| 12 | - | - | 37 | 1.19 | - | - | 36 | 3.90 | - | - |
| 13 | - | - | 77 | 15.18 | - | - | 78 | 9.89 | - | - |
| 14 | - | - | 74 | 18.58 | - | - | 83 | 13.36 | - | - |
| 15 | - | - | 64 | 17.16 | - | - | 68 | 10.08 | - | - |
| 16 | 39 | 580 | 49 | 7.19 | 41 | 4.64 | 46 | 2.78 | 40 | 1.73 |
| 17 | 43 | 277 | 51 | 9.97 | 43 | 3.91 | 50 | 2.60 | 43 | 3.38 |
| 18 | - | - | 56 | 21.53 | - | - | 54 | 15.76 | - | - |
| 19 | - | - | 53 | 12.85 | - | - | 57 | 8.01 | - | - |
| 20 | - | - | 101 | 20.42 | - | - | 95 | 13.58 | - | - |
| 21 | - | - | 73 | 16.16 | - | - | 69 | 10.47 | - | - |
| 22 | - | - | 94 | 18.45 | - | - | 87 | 14.54 | - | - |
| 23 | - | - | 72 | 13.95 | - | - | 71 | 10.81 | - | - |
| 24 | - | - | 79 | 14.18 | - | - | 75 | 9.50 | - | - |
| 25 | - | - | 73 | 14.81 | - | - | 66 | 9.03 | - | - |
| 26 | - | - | 60 | 14.23 | - | - | 61 | 9.35 | - | - |
| 27 | - | - | 81 | 15.31 | - | - | 80 | 12.72 | - | - |
| 28 | - | - | 87 | 27.15 | - | - | 89 | 23.74 | - | - |
| 29 | - | - | 51 | 21.58 | - | - | 52 | 16.70 | - | - |
| 30 | - | - | 59 | 13.41 | - | - | 59 | 9.61 | - | - |

Table 8.2: Solution lengths and computation times in solving Logistics with best-first search.

# Paper 9

# Planning with Pattern Databases

Stefan Edelkamp.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg

In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science, Springer, pages 13–24, 2001.

**Abstract**

Heuristic search planning effectively finds solutions for various benchmark planning problems, but since the estimates are either not admissible or too weak, optimal solutions are found in rare cases only. In contrast, heuristic pattern databases are known to significantly improve lower-bound estimates for optimally solving challenging single-agent problems like the 24-Puzzle and Rubik's Cube.

This paper studies the effect of pattern databases in the context of deterministic planning. Given a fixed state description based on instantiated predicates, we provide a general abstraction scheme to automatically create admissible domain-independent memory-based heuristics for planning problems, where abstractions are found in factorizing the planning space. We evaluate the impact of pattern database heuristics in A* and hill climbing algorithms for a collection of benchmark domains.

## 9.1 Introduction

General propositional planning is PSPACE complete [53], but when tackling specific benchmark planning instances, improving the solution quality usually reveals the intrinsic hardness of the problems. For example, plan existence of Logistic and Blocks World problem instances is polynomial, but minimizing the solution lengths for these planning problems is NP-hard [168]. On the other hand for some benchmark domains like *Sokoban* and *Mystery* even plan existence is NP-hard. Therefore, we propose a planner that is able to find optimal plans and, if challenging planning problems call for exponential resources, the planner approximates the optimal solution.

### 9.1.1 Optimal Planning Approaches

*Graphplan* [39] constructs a layered planning graph containing two types of nodes, action nodes and proposition nodes. In each layer the preconditions of all operators are matched, such that *Graphplan* considers instantiated actions at specific points in time. *Graphplan* generates partially ordered plans to exhibit concurrent actions and alternates between two phases: *graph extension* to increase the search depth and *solution extraction* to terminate the planning process. *Graphplan* finds optimal parallel plans, but does not approximate solution lengths; it simply exhausts the given resources.

Another optimal planning approach is symbolic exploration simulating a breadth-first search according to the binary encoding of planning states. The operators unfold the initial state over time and an efficient theorem prover then searches for a satisfying truth assignment. A Boolean formula $f_t$ describes the set of states reachable in $t$ steps. If $f_t$ contains a goal state, the problem is solvable with the minimal $t$ as the optimal solution length.

Two approaches have been proposed. *Satplan* [204] encodes the planning problem with a standard representation of Boolean formulae as a conjunct of clauses.

The alternative in the planner *Mips* [103] is to apply binary decision diagrams (BDDs); a data structure providing a unique representation for Boolean functions [50]. The BDD planning approach is in fact *reachability analysis* in model checking [65]. It applies to both deterministic and non-deterministic planning and the generated plans are optimal in the number of sequential execution steps. Usually, symbolic approaches cannot approximate except for recent preliminary results with domain abstractions [196] and with symbolic best-first search [114]. Though promising, the solution quality is not as good as in explicit search.

### 9.1.2 Heuristic Search Planning

Directed search is currently the most effective approach in classical AI-planning: four of five honored planning systems in the general planning track of the AIPS-2000 competition at least partially incorporate heuristic search. However, in traversing the huge state spaces of all combinations of grounded predicates, all planners rely on inadmissible estimates. The currently fastest deterministic planner, FF [181], solves a relaxed planning problem for each state to compute an inadmissible estimate. Furthermore, non-general pruning rules in FF such as *helpful action cuts* and *goal ordering cuts* help to avoid plateaus and local optima in the underlying hill-climbing algorithm. Completeness in undirected

problem graphs is achieved by breadth-first searching improvements for the estimate and by omitting pruning in case of backtracks. Nevertheless, the daunting problem for FF are directed problem graphs with dead-ends from which its move committing hill-climbing algorithm cannot recover.

The best admissible estimate that has been applied to planning is the *max-pair* heuristic [163] implemented in the HSP planner. However, even by sacrificing optimality due to scaling, in AIPS-2000 this estimate was too weak to compete with the FF-heuristic. Moreover, own experiments with an improvement to *max-pair* according to a minimum matching on a graph weighted with fact-pair solution lengths were discouraging.

This paper proposes a pre-computed admissible heuristic that easily outperforms *max-pair* and by scaling the influence of the heuristic even the state-of-the-art FF-heuristic is beaten. To build the database we exhaustively search all state-to-goal distances in tractable abstractions of the planning state-space that serve as lower bound estimates for the overall problem. After studying the pattern database framework, we present experiments with a sizable number of benchmark planning problems of AIPS-1998 and AIPS-2000 and draw concluding remarks.

## 9.2 Planning Space Representation

For the sake of simplicity we concentrate on the STRIPS formalism [126], in which each operator is defined by a precondition list $P$, an add list $A$, and a delete list $D$, but the presented approach can be extended to various problem description languages which can be parsed into a fixed state encoding. We refer to state descriptions and lists as sets/conjuncts of *grounded* predicates also called *facts* or *atoms*. This is not a limitation since all state-of-the-art planners perform grounding; either prior to the search or on the fly.

**Definition 2** *Let $F$ be the set of grounded predicates and $O$ be a set of grounded STRIPS operators. The result $S'$ of an operator $o = (P, A, D) \in O$ applied to a state $S \subseteq F$ is defined as $S' = (S \setminus D) \cup A$ in case $P \subseteq S$. Inverse STRIPS operators $o^{-1}$ are given by $o^{-1} = ((P \setminus D) \cup A, D, A)$.*

We exemplify our considerations in the Blocks World domain of AIPS-2000, specified with the four operators `pick-up`, `put-down`, `stack`, and `unstack`. For example, the grounded operator `(pick-up a)` is defined as

$P = \{$`(clear a)`, `(ontable a)`, `(handempty)`$\}$,
$A = \{$`(holding a)`$\}$, and
$D = \{$`(ontable a)`, `(clear a)`, `(handempty)`$\}$

The goal of the instance 4-1 is defined by $\{$`(on d c)`,`(on c a)`,`(on a b)`$\}$ and the initial state is given by $\{$`(clear b)` `(ontable d)`, `(on b c)`, `(on c a)`, `(on a d)`$\}$.

The first step to construct a pattern database is a domain analysis prior to the search. The output are *mutex groups* of mutually exclusive facts. In every state (reachable from the initial state), exactly one of the atoms in each group will be true. In general this construction is not unique such that we minimize the state description length over all possible partitionings as proposed for the MIPS planning system [101]. In the example problem we find the following nine mutex-groups.

- $G_1 = \{$`(on c a)`, `(on d a)`, `(on b a)`, `(clear a)`, `(holding a)`$\}$,
- $G_2 = \{$`(on a c)`, `(on d c)`, `(on b c)`, `(clear c)`, `(holding c)`$\}$,
- $G_3 = \{$`(on a d)`, `(on c d)`, `(on b d)`, `(clear d)`, `(holding d)`$\}$,
- $G_4 = \{$`(on a b)`, `(on c b)`, `(on d b)`, `(clear b)`, `(holding b)`$\}$,
- $G_5 = \{$`(ontable a)`, `true`$\}$,
- $G_6 = \{$`(ontable c)`, `true`$\}$,
- $G_7 = \{$`(ontable d)`, `true`$\}$,
- $G_8 = \{$`(ontable b)`, `true`$\}$, and
- $G_9 = \{$`(handempty)`, `true`$\}$,

where `true` refers to the situation, when none of the other atoms is present in a given state description.

**Definition 3** *Let $G = \{G_1, \ldots, G_k\}$ with $G_i \subseteq F \cup \{\texttt{true}\}$ for $i \in \{1, \ldots, k\}$ be the set of* mutex groups*, i.e. $f_i \neq f_j$ for $f_i \in G_i \setminus \{\texttt{true}\}$ and $f_j \in G_j \setminus \{\texttt{true}\}$. A* state *is a conjunct $f_1 \wedge \ldots \wedge f_k$ of facts $f_i \in G_i$, $i \in \{1, \ldots, k\}$. All represented states span the planning space*[1] $\mathcal{P}$.

## 9.3 Pattern Databases

A recent trend in single-agent search is to calculate the estimate with heuristic pattern databases (PDBs) [75]. The idea is to generate heuristics that are defined by distances in space abstractions. PDB heuristics are consistent[2] and have been effectively applied to solve challenging $(n^2 - 1)$-Puzzles [218] and Rubik's Cube [215]. In the $(n^2 - 1)$-Puzzle a pattern is a collection of tiles and in Rubik's Cube either a set of edge-*cubies* or a set of corner-*cubies* is selected.

For all of these problems the construction of the PDB has been implemented problem-dependently, i.e. by manual input of the abstraction for the puzzles and its storage by suitable perfect hash functions. In contrast, we apply the concept of PDBs to general problem-independent planning and construct pattern databases fully automatically.

[1] The planning space $\mathcal{P}$ is in fact smaller than the set of subsets of grounded predicates, but includes the set of states reachable from the initial state.

[2] Consistent heuristic estimates fulfill $h(v) - h(u) + w(u, v) \geq 0$ for each edge $(u, v)$ in the underlying, possibly weighted, problem graph. They yield monotone merits $f(u) = g(u) + h(u)$ on generating paths with weight $g(u)$. Admissible heuristics are lower bound estimates which underestimate the goal distance for each state. Consistent estimates are indeed admissible.

### 9.3.1 State Abstractions

State space abstractions in the context of PDBs are concisely introduced in [183]: A state is a vector of fixed length and operators are conveniently expressed by label sets, e.g. an operator mapping $\langle A, B, _\rangle$ to $\langle B, A, _\rangle$ corresponds to a transposition of the first two elements for any state vector of length three. The state space is the transitive closure of the seed state $S_0$ and the operators $O$. A *domain abstraction* is defined as a mapping $\phi$ from one label set $L$ to another label set $K$ with $|K| < |L|$ such that states and operators can be simplified by reducing the underlying label set. A *state space abstraction* of the search problem $\langle S_0, O, L\rangle$ is denoted as $\langle \phi(S_0), \phi(O), K\rangle$. In particular, the abstraction mapping is non-injective such that the abstract space (which is the image of the original state space) is therefore much smaller than the original space.

The framework in [183] only applies to certain kinds of permutation groups, where in our case the abstract space is obtained in a more general way, since abstraction is achieved by projecting the state representation.

**Definition 4** *Let $F$ be the set of grounded predicates. A* planning space abstraction *$\phi$ is a mapping from $F$ to $F \cup \{\texttt{true}\}$ such that for each group $G$ either for all $f \in G: \phi(f) = f$ or for all $f \in G: \phi(f) = \texttt{true}$.*

Since planning states are interpreted as conjuncts of facts, $\phi$ can be extended to map each planning state of the original space $\mathcal{P}$ to one in the abstract space $\mathcal{A}$. In the example problem instance we apply two planning space abstractions $\phi_{odd}$ and $\phi_{even}$. The mapping $\phi_{odd}$ assigns all atoms in groups of odd index to the trivial value `true` and, analogously, $\phi_{even}$ maps all fluents in groups with even index value to `true`. All groups not containing a atoms in the goal state are also mapped to `true`[3]. In the example, the goal is partitioned into $\phi_{even}(G) = \{\texttt{(on c a)}\}$ and $\phi_{odd}(G) = \{\texttt{(on a b)}, \texttt{(on d c)}\}$, since the groups $G_4$ to $G_9$ are not present in the goal description.

Abstract operators are defined by intersecting their precondition, add and delete lists with the set of non-reduced facts in the abstraction. This accelerates the construction of the pattern table, since several operators simplify.

**Definition 5** *Let $\phi$ be a planning space abstraction and $\delta_\phi(S_1, S_2)$ be the graph-theoretical shortest path between to two states $S_1$ and $S_2$ in $\mathcal{A}$. Furthermore, let $S_0$ be the start and $S_t$ be the set of goal states in $\mathcal{P}$. A* planning pattern database *(PDB) according to $\phi$ is a set of pairs, with the first component being an abstract planning state $S$ and the second component being the minimal solution length in the abstract problem space, i.e.,*

$$\text{PDB}(\phi) = \{(S, \delta_\phi(S, \phi(S_t))) \mid S \in \mathcal{A}\}.$$

*PDB*($\phi$) is calculated in a breadth-first traversal starting from the set of goals in applying the inverse of the operators. Two facts about PDBs are important. When reducing the state description length $n$ to $\alpha n$ with $0 < \alpha < 1$ the state space and the search tree shrinks exponentially; e.g. $2^n$ bit vectors correspond to an abstract space of $2^{\alpha n}$ elements.

The second observation is that once the pattern database is calculated, accessing the heuristic estimate is fast by a simple table lookup (cf. Section 9.3.3). Moreover, PDBs

[3] To include mutex-groups into PDB calculations which are not present in the goal state, we may generate *all possible instances* for the fact set. In fact, this is the approach that is applied in our implementation.

| ((clear a),1) |
|---|
| ((holding a),2) |
| ((on b a),2) |
| ((on d a),2) |

| ((on d c)(clear b),1) | ((on a b)(clear c),1) |
|---|---|
| ((on d c)(holding b),2) | ((clear c)(clear b),2) |
| ((on d c)(on d b),2) | ((on a b)(holding c),2) |
| ((on a c)(on a b) ,2) | ((clear c)(holding b),3) |
| ((clear b)(holding c),3) | ((on a c)(clear b),3) |
| ((on d b)(clear c),3) | ((holding c)(holding b),4) |
| ((on b c)(clear b),4) | ((on a c)(holding b),4) |
| ((on c b)(clear c),4) | ((on d b)(holding c),4) |
| ((on a c)(on d b),4) | ((on b c)(holding b),5) |
| ((on a b)(on b c),5) | ((on d b)(on b c),5) |
| ((on c b)(holding c),5) | ((on a c)(on c b),5) |
| ((on c b)(on d c),5) | |

Table 9.1: Pattern databases $PDB(\phi_{even})$ and $PDB(\phi_{odd})$ for the example problem.

can be re-used for the case of different initial states. $PDB(\phi_{even})$ and $PDB(\phi_{odd})$ according to the abstractions $\phi_{even}$ and $\phi_{odd}$ of our example problem are depicted in Table 9.1. Note that there are only three atoms present in the goal state such that one of the pattern databases only contains patterns of length one. Abstraction $\phi_{even}$ corresponds to $G_1$ and $\phi_{odd}$ corresponds to the union of $G_2$ and $G_4$.

### 9.3.2 Disjoint Pattern Databases

Disjoint pattern databases add estimates according to different abstractions such that the accumulated estimates still provide a lower bound heuristic.

**Definition 6** *Two pattern databases* PDB($\phi_1$) *and* PDB($\phi_2$) *are* disjoint, *if the sum of respective heuristic estimates always underestimates the overall solution length, i.e.,*

$$\delta_{\phi_1}(\phi_1(S), \phi_1(S_t)) + \delta_{\phi_2}(\phi_2(S), \phi_2(S_t)) \leq \delta(S, S_t), \forall S \in \mathcal{P}.$$

PDBs are not always disjoint. Suppose that a goal contains two atoms $p_1$ and $p_2$, which are in groups 1 and 2, respectively, and that an operator $o$ makes both $p_1$ and $p_2$ true. Then, the distance under abstraction $\phi_1$ is 1 (because the abstraction of $o$ will make $p_2$ in group 2 true in one step) and the distance under $\phi_2$ is also 1 (for the same reason). But the distance in the original search space is also 1.

**Definition 7** *An* independent abstraction set $I$ *is a set of group indices such that no operator affects both atoms in groups in* $I$ *and atoms in groups that are not in* $I$. *The according abstraction* $\phi_I$ *that maps all atom groups not in* $I$ *to true is called an* independent abstraction.

**Theorem 3** *A partition of the groups into independent abstractions sets yields disjoint pattern databases.*

**Proof:** Each operator changes information only within groups of a given partition and an operator of the abstract planning space contributes one to the overall estimate only if it changes facts in available fact groups. Therefore, by adding the solution lengths of different abstract spaces each operator on each path is counted at most once. ■

For some domains like Logistics operators act locally according to any partition into groups so that the precondition of Theorem 3 is trivially fulfilled.

### 9.3.3 Perfect Hashing

PDBs are implemented as a (perfect) hash table with a table lookup in time linear to the abstract state description length.

According to the partition into groups a perfect hashing function is defined as follows. Let $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ be the selected groups in the current abstraction and $\textit{offset}(k)$ be defined as $\textit{offset}(k) = \prod_{l=1}^{k} |G_{i_{l-1}}|$ with $|G_{i_0}| = 1$. Furthermore, let $\textit{group}(f)$ and $\textit{position}(f)$ be the group index and the position in the group of fact $f$, respectively. Then the perfect hash value $\textit{hash}(S)$ of state $S$ is

$$\textit{hash}(S) = \sum_{f \in S} \textit{position}(f) \cdot \textit{offset}(\textit{group}(f)).$$

Since perfect hashing uniquely determines an address for the state $S$, $S$ can be reconstructed given $\textit{hash}(S)$ by extracting all corresponding group and position information that define the facts in $S$. Therefore, we establish a good compression ratio, since each state in the queue for the breadth-first search traversal only consumes one integer. The breadth-first-search queue is only needed for construction and the resulting PDB is a plain integer array of size $\textit{offset}(k+1)$ encoding the distance values for the corresponding states, initialized with $\infty$ for patterns that are not encountered. Some states are not generated, since they are not reachable, but the above scheme is more time and space efficient than ordinary hashing storing the uncompressed state representation. Since small integer elements consume only a few bytes, on current machines we may generate PDBs of a hundred million entries and more.

### 9.3.4 Clustering

In the simple example planning problem the combined sizes of groups and the total size of the generated pattern databases $\textit{PDB}(\phi_{even})$ and $\textit{PDB}(\phi_{odd})$ differ considerably. Since we perform a complete exploration in the generation process, in larger examples the requirements in time and space resources for computing PDBs might be exhausted. Therefore, an automatic way to find a suitable balanced partition according to given memory limitations is required. Instead of a bound on the total size of all PDBs together, we globally limit the size of each pattern database, which is in fact the number of expected states. The restriction is not crucial, since the number of different pattern databases is small in practice. The threshold is the parameter to tune the quality of the estimate. Obviously, large threshold values yield optimal estimates in small problem spaces.

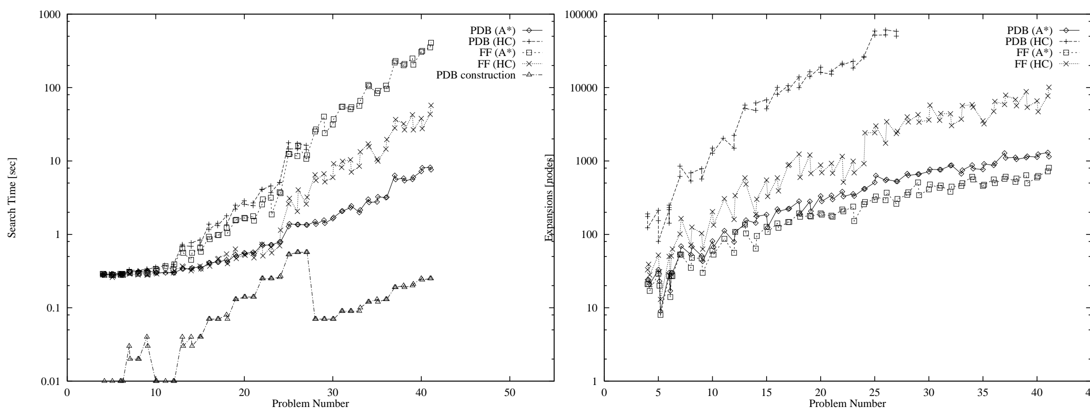


Figure 9.1: Time performances and numbers of expansions of A* and hill climbing in the Logistics domain with respect to the PDB and FF heuristic on a logarithmic scale. PDB construction time is included in the overall search time.

We are confronted with a Bin-Packing variant: Given the sizes of groups, the task is to find the minimal number of pattern databases such that the sizes do not exceed a certain threshold value. Notice that the group sizes are multiplied in order to estimate the search space size. However, the corresponding encoding lengths add up. Bin-Packing is NP-hard in general, but good approximation algorithms exist. In our experiments we applied the best-fit strategy.

## 9.4 Results

All experimental results were produced on a Linux PC, Pentium III CPU with 800 MHz and 512 MByte. We chose the most efficient domain-independent planners as competitors. In Logistics, the program FF is chosen for comparison and in Blocks World, the pattern database approach is compared to the optimal planner *Mips*.

### 9.4.1 Logistics

We applied PDBs to Logistics and solved the entire problem set of AIPS-2000. The largest problem instance includes 14 trucks located in one of three locations of the 14 cities. Together with four airplanes the resulting state space has a size of about $3^{14} \cdot 14^4 \cdot 60^{42} \approx 8.84223 \cdot 10^{85}$ states. All competing planners that have solved the entire benchmark problem suite are (enforced) hill-climbers with a variant of the FF heuristic and the achieved results have about the same characteristics [178]. Therefore, in Table 9.1 we compare the PDB approach with the FF-heuristic. In the enforced hill climbing algorithm we allow both planners to apply branching cuts, while in A* we scale the influence of the heuristic with a factor of two. The effects of scaling are well-known [287]: weightening A* possibly results in non-optimal solution, but the search tends to succeed much faster. In the AIPS-2000 competition, the scaling factor 2 has enhanced the influence of the *max-pair* heuristic in the planner HSP. However, even with this improvement it solves only a few problems of this benchmark suite.

The characteristics of the PDB and FF heuristics in Figure 9.1 are quite different. The number of expanded nodes is usually larger for the former one but the run time is

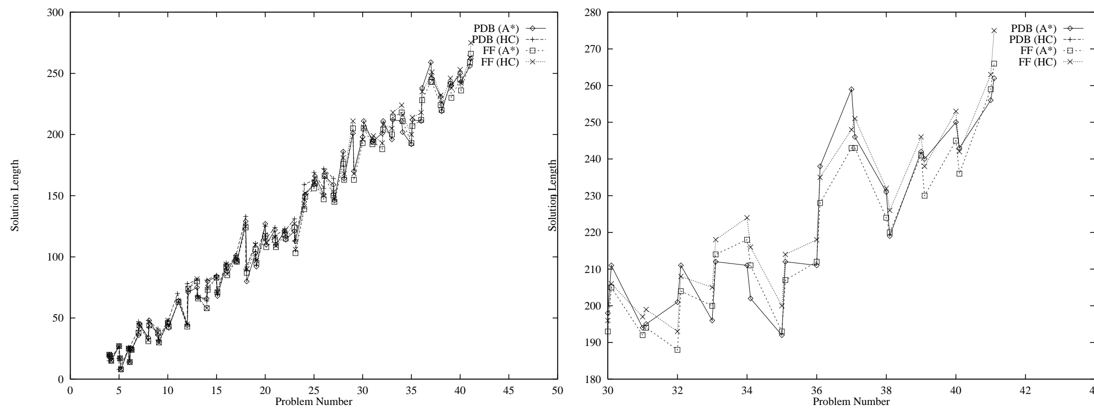


Figure 9.2: Overall and magnified solution quality of A* and enforced hill climbing in the Logistics domain with respect to to the PDB and FF heuristic.

much shorter. A* search with PDBs outperforms FF with hill climbing *and* branching cuts. The savings are about two orders of magnitude with respect to FF and A* and one order of magnitude with respect to FF and hill climbing, while the effect for the number of expansions is the exact opposite. In the example set the average time for a node expansion in PDB-based planning is smaller by about two orders of magnitude compared to FF.

On the other hand, in larger problem instances enforced hill climbing according to the PDB heuristic generates too many nodes to be kept in main memory. In a few seconds the entire memory resources were exhausted. This suggests applying memory limited search algorithm like thresholding in IDA* and alternative hashing strategies to detect move transpositions in high search depths.

We summarize that hill climbing is better suited to the FF heuristic while weighted A* seems to perform better with PDBs. The solution qualities are about the same as Figure 9.2 deptics. Even magnification to larger problem instances fails to establish a clear-cut winner.

**Blocks World** Achieving approximate solutions in Blocks World is easy; 2-approximations run in linear time [327]. Moreover, different domain-dependent cuts drastically reduce the search space. Hence, TALPlanner [229] with hand-coded cuts and FF with hill climbing, helpful action and goal ordering cuts find good approximate solutions to problems with fifty Blocks and more.

FF using enforced hill climbing without cuts is misguided by its heuristic, backtracks and tends to get lost in local optima far away from the goal. We concentrate on optimal solutions for this domain. Since any $n$-Tower configuration is reachable from the initial state state, the state space grows exponentially in $n$, and indeed, optimizing Blocks World is NP-hard. *Graphplan* is bounded to about 9 blocks and no optimal heuristic search engine achieves a better performance, e.g. HSP with *max-pair* is bounded to about 6-7 blocks. Model checking engines like BDD exploration in *Mips* and iterative Boolean satisfiability checks in *Satplan* are best in this domain and optimally solve problems with up to 12-13 blocks. Table 9.3 depicts that PDBs are competitive and that the solution lengths match.

Moreover, better scaling in time seems to favor PDB exploration. However, in both approaches space consumption is more crucial than time. In the bidirectional symbolic

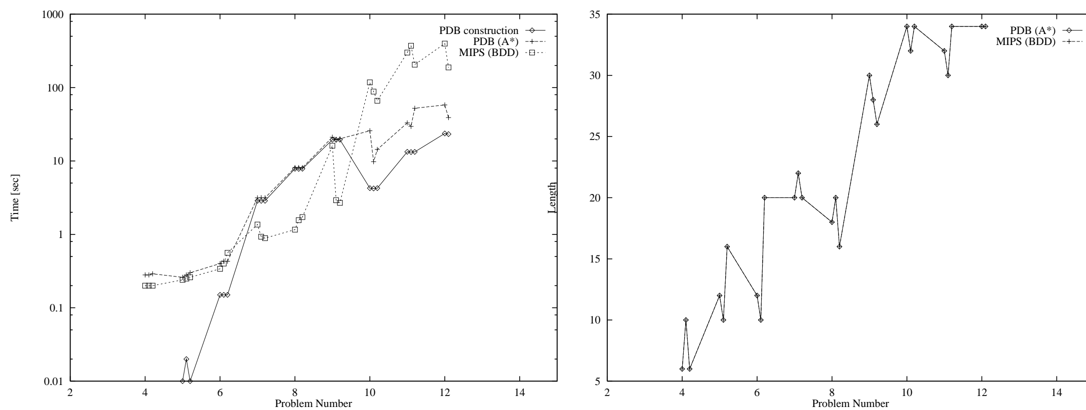


Figure 9.3: Time performance and solution quality of BDD expoloration and optimal PDB planning in Blocks World. PDB construction time is included in the overall search time.

breadth-first search engine of Mips the BDD sizes grow very rapidly and large pattern databases with millions of entries still lead to millions of node expansions. When searching for optimal solutions to 13-block benchmark problems this thrashes the memory resources in both planning approaches. In summary, optimal solving Blocks World is still hard for general planning engines.

### 9.4.2 Other Domains

*Gripper* (AIPS-1998) spans an exponentially large but well-structured search space such that greedy search engines finds optimal solutions. On the other hand, Gripper is known to be hard for *Graphplan*. Both FF with hill-climbing and cuts and PDB with weighted A* find optimal solutions in less than a second.

Like Logistics, the NP-hard [168] *Mystery* domain (AIPS-1998) is a transportation domain on a road map. Trucks are moving around this map and packages are being carried by the mobiles. Additionally, various *capacity* and *fuel constraints* have to be satisfied. Mystery is particularly difficult for heuristic search planning, since some of the instances contain a very high portion of undetected dead-ends [178]. In contrast to the most effective heuristic search planner GRT [302], the PDB planning algorithm does not yet incorporate manual reformulation based on explicit representation of resources. However, experiments show that PDB search is competitive: problems 1-3, 9, 11, 17, 19, 25-30 were optimally solved in less then 10 seconds, while problem 15 and 20 required about 5 and 2 minutes, respectively. At least problem 4,7, and 12 are not solvable. Time performance and the solution qualities are better than in [302] Scaling reduces the number of node expansion in some cases but has not solved any new problem.

The start position of *Sokoban* consists of a selection of balls within a maze and a designated goal area into which the balls have to be moved. A man, controlled by the puzzle solver, can traverse the board and push balls onto adjacent empty squares. Sokoban has been proven to be PSPACE complete and spans a directed search space with exponentially many dead-ends, in which some balls cannot be placed onto any goal field [199]. Therefore, hill climbing will eventually encounter a dead-end and fail. Only overall search schemes like A*, IDA* or best-first prevent the algorithm from getting trapped. In our experiments we optimally solved all 52 automatically generated problems [274] in less than

five seconds each. The screens were compiled to PDDL with a one-to-one ball-to-goal mapping so that some problems become unsolvable. Since A* is complete we correctly establish unsolvability of 15 problems in the test set. Note that the instances span state spaces much smaller than the 90 problem suite considered in [199] with problems currently too difficult to be solved with domain independent planning.

As expected, additional results in Sokoban highlight that in contrast to the PDB-heuristic, the FF-heuristic, once embedded in A*, yields good but not optimal solutions. BDD exploration in Mips does find optimal solutions, but for some instances it requires over a hundred seconds for completion.

## 9.5 Conclusion

Heuristic search is currently the most promising approach to tackle huge problem spaces but usually does not yield optimal solutions. The aim of this paper is to apply recent progress of heuristic search in finding optimal solutions to planning problems by devising an automatic abstraction scheme to construct pre-compiled pattern databases.

Our experiments show that pattern database heuristics are very good admissible estimators. Once calculated our new estimate will be available in constant time since the estimate of a state is simply retrieved in a perfect hash table by projecting the state encoding. We will investigate different pruning techniques to reduce the large branching factors in planning. There are some known general pruning techniques such as *FSM pruning* [337], *Relevance Cuts* and *Pattern Searches* [199] that should be addressed in the near future.

Although PDB heuristics are admissible and calculated beforehand, their quality can compete with the inadmissible FF-heuristic that solves a relaxed planning problem for *every* expanded state. The estimates are available in a simple table lookup, and, in contrast to the FF-heuristic, A* finds optimal solutions. Weighting the estimate helps to cope with difficult instances for approximate solutions. Moreover, PDB heuristics in A* can handle directed problem spaces and prove unsolvability results.

One further important advantage of PDB heuristics is the possibility of a symbolic implementation. Due to the representational expressiveness of BDDs, a breadth-first search (BFS) construction can be completed with respect to larger parts of the planning space for a better quality of the estimate. The exploration yields a relation $H(\textit{estimate}, \textit{state})$ represented with a set of Boolean variables encoding the BFS-level and a set of variables encoding the state. Algorithm BDDA*, a symbolic version of A*, integrates the symbolic representation of the estimate [114]. Since PDBs lead to consistent heuristics the number of iterations in the BDDA* algorithms is bounded by the square of the solution length. Moreover, symbolic PDBs can also be applied to explicit search. The heuristic estimate for a state can be determined in time linear to the encoding length.

# Paper 10

# Symbolic Pattern Databases in Heuristic Search Planning

Stefan Edelkamp.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg

In *Artificial Intelligence Planning and Scheduling*, Lecture Notes in Computer Science, Springer, 2002.

**Abstract**

This paper invents symbolic pattern databases (SPDB) to combine two influencing aspects for recent progress in domain-independent action planning, namely heuristic search and model checking. SPDBs are off-line computed dictionaries, generated in symbolic backward traversals of automatically inferred planning space abstractions.

The entries of SPDBs serve as heuristic estimates to accelerate explicit and symbolic, approximate and optimal heuristic search planners. Selected experiments highlight that the symbolic representation yields much larger and more accurate pattern databases than the ones generated with explicit methods.

## 10.1 Introduction

Heuristic search is one of the most effective search techniques to cope with very large problem spaces. The guidance for search algorithms like A* [161] and IDA* [213] are estimators that approximate the remaining distance to the goal.

The additional information focuses the search into the direction of the goal and its quality mainly influences the number of states to be expanded; the better the estimate, the larger the reduction in search efforts.

Planning problems implicitly generate weighted problem graphs by applying operator sequences to their seed states. By changing operator costs, A* can be casted as a variant of Dijkstra's single-source shortest path algorithm: the new costs of the operators are set to the old ones minus the heuristic value of the expanded state, plus the estimate of the successor state [115]. Admissible heuristics are lower bound problem relaxations, yield optimal solutions, but may introduce negative weights calling for re-openings of already expanded states [287].

Pattern databases (PDBs) are dictionaries of heuristic values that have been originally applied to the Fifteen Puzzle [75]. In this context, PDBs are generalizations of the Manhattan distance heuristic, that corresponds to subproblem solutions of moving each tile onto its goal position. The PDB representation is a selection of look-up tables memorizing the goal distances of each tile at any board location. Since the subproblems are independent (only one tile can move at a time), the minimum numbers of moves to solve the individual puzzles can be added; still providing an admissible heuristic. Refined PDBs take not only one but a selection of interacting tiles (the pattern) into account. A large hash table stores their combined goal distances on a simplified board with all other tiles removed. PDBs are generated in complete backward explorations, starting from the set of abstract goals.

The PDB approach has been extended to find first optimal solutions to random Rubik's Cube problems [213], where a pattern corresponds to a selection of side or corner cubies. Independence of PDBs has been exploited to solve the 24-Puzzle [218]. In all cases the abstractions for PDB construction were hand-tailored and domain dependent. The effectiveness of PDBs in form of a space-time trade-off reveals that PDBs size is inversely correlated to the resulting search effort [183].

Steps towards the automated creation of PDB heuristics base on local search in the space of PDBs [173] and change the abstraction level according to the predicted search efforts. However, the approach is currently limited to moderate state-space sizes, or restricted to easier exploration tasks like the computation of macro operators.

Explicit PDB heuristics that have been proposed for domain-independent action planning [95] share similarities with PDBs in sliding-tile puzzles and challenge even on-line computed estimates like the FF-heuristic [180]. The rough idea is to interpret the set propositional atoms as tiles, so that a planning pattern is a selection of them. The approach first infers groups of mutually exclusive facts. In every reachable state exactly one of the atoms in each group is true. The group information is exploited to derive planning abstractions and to infer perfect hash functions for pattern storage. Automated clustering partitions the state space into a set of abstractions with state spaces that fit into main memory. Planning PDBs are not always independent, but suitable partitions into groups, where all operators affect only atoms in the specified set, always yield independent PDBs.

In this paper we propose symbolic pattern databases (SPDBs) instead of explicit ones.

A SPDB is Boolean function $H$ of tuples $(v, S)$. For a given planning state $S$ with value $v$, formula $H$ evaluates to *true* if and only if the estimate of $S$ equals $v$. Through an efficient representation of Boolean functions, larger PDBs and more accurate estimates can be inferred and utilized.

The structure of the paper is as follows. First we give a concise introduction to PDBs together with proofs of some important properties. Then we turn the to symbolic representation of planning states and operators. Next we introduce symbolic backward exploration to generate SPDBs and integrate this representation of the heuristic estimate into directed explicit and symbolic search search engines. This algorithmic treatment is followed by a discussion the influence of SPDBs to search tree growth and exploration efforts. We evaluate the impact of the algorithms, taking Blocks World as a selected case study. Finally, we discuss related work, and finish with a few concluding remarks.

## 10.2 Pattern Databases in AI-Planning

For the sake for simplicity, throughout the paper we consider grounded propositional planning problems in STRIPS notation [126] and stick to sequential plan generation. However, the framework also applies to more general planning domain description languages [131].

### 10.2.1 Grounded Propositional Planning

Most successful planners perform grounding.

**Definition 1** *A* grounded propositional planning problem *is a finite state space problem* $\mathcal{P} = <\mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G}>$, *where* $\mathcal{S} \subseteq 2^A$ *is the set of states,* $2^A$ *is the power set of set of propositions* $A$, $\mathcal{I} \in \mathcal{S}$ *is the initial state,* $\mathcal{G} \subseteq \mathcal{S}$ *is the set of goal states, and* $\mathcal{O}$ *is the set of operators that transform states into states. Operators* $o = (\alpha, \beta) \in \mathcal{O}$ *have propositional preconditions* $\alpha$, *and propositional effects* $\beta = (\beta_a, \beta_d)$, *where* $\alpha \subseteq A$ *is the precondition list,* $\beta_a \subseteq A$ *is the add list and* $\beta_d \subseteq A$ *is the delete list. Given a state* $S$ *with* $\alpha \subseteq S$ *then its successor is* $S' = S \cup \beta_a \setminus \beta_d$.

Sequential plans are defined as follows.

**Definition 2** *A* sequential plan $\pi = (O_1, \ldots, O_k)$ *is an ordered sequence of operators* $O_i \in \mathcal{O}$, $i \in \{1, \ldots, k\}$, *that transforms the initial state* $\mathcal{I}$ *into one of the goal states* $G \in \mathcal{G}$, *i.e. there exists a sequence of states* $S_i \in \mathcal{S}$, $i \in \{0, \ldots, k\}$, *with* $S_0 = \mathcal{I}$, $S_k = G$ *and* $S_i$ *is the outcome of applying* $O_i$ *to* $S_{i-1}$, $i \in \{1, \ldots, k\}$. *The length of a plan* $\pi$ *is* $k$, *and the minimal* $k$ *is the optimal sequential plan length* $\delta(\mathcal{I})$.

### 10.2.2 Abstract Planning problems

Instead of the PDB definition based on fact groups [95], in this paper we prefer a formal treatment that directly builds upon the above state space characterization.

**Definition 3** *Let* $R \subseteq A$ *be a set of propositional atoms. A* restriction $\phi_R$ *is a mapping from* $2^A$ *into* $2^A$ *defined as* $\phi_R(P) = \{a \in P \mid a \in R\}$. *For* $\phi_R(P)$ *we also write* $P|_R$.

Restrictions imply planning problem abstractions.

**Definition 4** *An* abstract planning problem $\mathcal{P}|_R = \; < \mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R >$ *of a grounded propositional planning problem* $< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ *with respect to a set of propositional atoms* $R$ *is defined by*

1. $\mathcal{S}|_R = \{\phi_R(S) \mid S \in \mathcal{S}\}$,
2. $\mathcal{I}|_R = \phi_R(\mathcal{I})$,
3. $\mathcal{G}|_R = \{\phi_R(G) \mid G \in \mathcal{G}\}$,
4. $\mathcal{O}|_R = \{\phi_R(O) \mid O \in \mathcal{O}\}$, *with* $\phi_R(O)$ *for* $O = (\alpha, (\beta_a, \beta_d)) \in \mathcal{O}$ *defined as* $\phi_R(O) = (\alpha|_R, (\beta_a|_R, \beta_d|_R))$.

*Sequential abstract plans for the abstract planning problem* $\mathcal{P}|_R$ *are denoted by* $\pi_R$ *and optimal abstract sequential plan length is denoted by* $\delta_R$.

Abstract operators are fixed by intersecting their precondition, add and delete lists with the set of non-reduced facts in the abstraction. Restriction of operators in the original space may yield *void* operators $\phi_R(O) = (\emptyset, (\emptyset, \emptyset))$ in the abstract planning problem, which are discarded from the operator set $\mathcal{O}|_R$.

The next result shows that our definition of abstraction is sound.

**Lemma 2** *Let* $R$ *be a set of propositional atoms. Restriction* $\phi_R$ *is* solution preserving, *i.e., for any sequential plan* $\pi$ *for the grounded propositional planning problem* $\mathcal{P} = \; < \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ *there exists a sequential plan* $\pi_R$ *for the planning state abstraction* $\mathcal{P}|_R = \; < \mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R >$.

*Moreover, an optimal sequential abstract plan* $\pi_R^{opt}$ *for* $\mathcal{P}|_R$ *is always shorter than an optimal sequential plan* $\pi^{opt}$ *for* $\mathcal{P}$, *i.e.* $\delta_R(S|_R) \leq \delta(S)$, *for all* $S \in \mathcal{S}$.

**Proof:** Let $\pi = (O_1, \ldots, O_k)$ be a sequential plan for $< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$. Then $\pi|_R = (O_1|_R, \ldots, O_k|_R)$ is a solution for $\mathcal{P}|_R = \; < \mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R >$.

Now suppose, that $\delta_R(S|_R) > \delta(S)$ for some $S \in \mathcal{S}$ and let $\pi^{opt} = (O_1, \ldots, O_t)$ be the optimal sequential plan from $S$ to $\mathcal{G}$ in the original planning space $\mathcal{P}$ then $\pi^{opt}|_R = (O_1|_R, \ldots, O_t|_R)$ is a valid plan in $\mathcal{P}|_R$ with plan length less or equal to $t = \delta(S)$. This is a contradiction to our assumption. ■

Strict inequality $\delta_R(S|_R) < \delta(S)$ is given if some operators $O_i|_R$ are void, or if there are alternative even shorter paths in the abstract space.

### 10.2.3 Planning Pattern Databases

The above setting allows a precise characterization of planning PDBs.

**Definition 5** *A planning PDB* $\mathcal{PDB}_R$ *with respect to a set of propositions* $R$ *and a grounded propositional planning problem* $< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ *is a collection of pairs* $(v, S)$ *with* $v \in \mathbb{N}$ *and* $S \in \mathcal{S}|_R$, *such that* $v = \delta_R(S)$. *Therefore,*

$$\mathcal{PDB}_R = \{(\delta_R(S), S) \mid S \in \mathcal{S}|_R\}.$$

In other words, a PDB is look-up table, addressed by the abstract planning state providing its minimal abstract solution length.

### 10.2.4 Disjoint Pattern Databases

Disjoint PDBs are important to derive admissible estimates.

**Definition 6** *Two planning PDBs $\mathcal{PDB}_R$ and $\mathcal{PDB}_Q$ with respect to $R, Q \subseteq A$, $R \cap Q = \emptyset$ are* disjoint*, if for all $O' \in \mathcal{O}|_R$, $O'' \in \mathcal{O}|_Q$ we have $\phi_R^{-1}(O') \cap \phi_Q^{-1}(O'') = \emptyset$, where $\phi_R^{-1}(O') = \{O \in \mathcal{O} \mid \phi_R(O) = O'\}$.*

**Lemma 3** *Two disjoint planning PDBs $\mathcal{PDB}_R$ and $\mathcal{PDB}_Q$ for a grounded propositional planning problem $< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ and sets of propositions $P$ and $Q$ are* additive*: for all $S \in \mathcal{S}$ we have $\delta_P(S|_R) + \delta_Q(S|_Q) \leq \delta(S)$.*

**Proof:**
Let $< \mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R >$ and $< \mathcal{S}|_Q, \mathcal{O}|_Q, \mathcal{I}|_Q, \mathcal{G}|_Q >$ be abstractions of $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ according to $P$ and $Q$, respectively, and let $\pi = (O_1, \ldots, O_k)$ be an optimal sequential plan for $\mathcal{P}$. Then, the abstracted plan $\pi|_R = (O_1|_R, \ldots, O_k|_R)$ is a solution for the state space problem $< \mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R >$ and $\pi|_Q = (O_1|_Q, \ldots, O_k|_Q)$ is a plan for $< \mathcal{S}|_Q, \mathcal{O}|_Q, \mathcal{I}|_Q, \mathcal{G}|_Q >$. We assume that all void operators in $\pi|_Q$ and $\pi|_R$, if any, are removed. Let $k_Q$ and $k_R$ be the resulting respective lengths of $\pi|_Q$ and $\pi|_R$.

Since the PDBs $\mathcal{PDB}_R$ and $\mathcal{PDB}_Q$ are disjoint, $O' \in \pi|_R$ and all $O'' \in \pi|_Q$ we have $\phi_R^{-1}(O') \cap \phi_Q^{-1}(O'') = \emptyset$. Therefore, $\delta_R(S|_R) + \delta_Q(S|_Q) \leq k_R + k_Q \leq \delta(S)$. ■

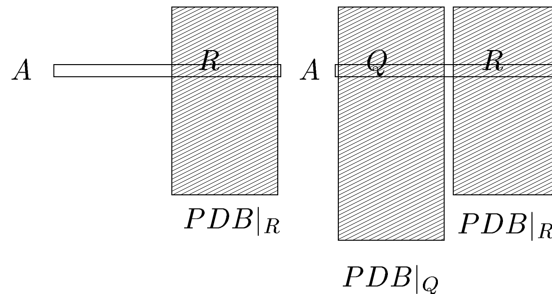


Figure 10.1: Illustration of a Planning PDB $\mathcal{PDB}_R$ and Two Disjoint Planning PDBs $\mathcal{PDB}_R$ and $\mathcal{PDB}_Q$.

In Figure 10.1 we have illustrated (disjoint) planning PDBs with respect to the given underlying set $A$ of propositions to encode a state.

In practice some operators remain non-void in different abstraction. For example, in our abstractions Logistics always yields disjoint PDBs, while in Blocks World some interdependent operators remain, since operators in Logistics modify either the location of a package, a truck or an airplane without affecting the others, while in Blocks World a *stack* operation changes both the status of the hand and the block.

In order to retain admissible estimates for the latter case, during construction conflicting operators can be assigned to cost zero for all but one PDB. Nevertheless, this technique of enforced *admissibility* may reduce the quality of the inferred estimate.

### 10.2.5 Partitions and Storage

Next, we consider sets of pattern databases.

**Definition 7** *A* partition $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ *is a collection of propositional sets* $Q_i$, $i \in \{1, \ldots, k\}$, *with* $Q_i \cap Q_j = \emptyset$, $1 \leq i < j \leq k$ *and* $\bigcup_{i=1}^{k} Q_i = A$. *A* planning space partition $\mathcal{P}_\mathcal{Q}$ *according to a partition* $\mathcal{Q}$ *is a collection of planning problems* $< \mathcal{S}|_{Q_i}, \mathcal{O}|_{Q_i}, \mathcal{I}|_{Q_i}, \mathcal{G}|_{Q_i} >$, $Q_i \subseteq \mathcal{Q}$, $i \in \{1, \ldots, k\}$.

The following result is an immediate generalization of Lemma 3.

**Lemma 4** *Pairwise disjoint planning PDBs according to planning space partition* $\mathcal{P}_\mathcal{Q}$ *for a grounded propositional planning problem* $< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ *and a* partition $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ *are* additive*, i.e. for all* $S \in \mathcal{S}$ *we have* $\delta_{Q_1}(S|_{Q_1}) + \ldots + \delta_{Q_k}(S|_{Q_k}) \leq \delta(S)$.

Finding a suitable partition that leads to pairwise disjoint or to almost pairwise disjoint planning PDBs is not trivial. For two databases the task is a variant of the graph partitioning problem (GPP), which divides the vertices of a given graph into two equally sized subsets, so that the number of edges from one subset the other one is minimized. In our setting, vertices correspond to atoms and edges to operators. Since GPP is NP complete and the number of atoms is considerably large, we mimic the approach of [95], which simplifies the problem of finding a suitable partition of the set of facts to a form of bin-packing (BPP). For this case, interdependencies are neglected. A group can be added to an already existing abstraction, if the combined state space still fits into main memory. BPP is NP complete but has several efficient approximation algorithms. Currently, we study how goal fact dependencies can improve the established partition.

In explicit PDB construction, the PDBs themselves and the transposition tables [305] are represented as hash tables. Therefore, the limit for PDB construction is the number of (abstract) states that can be hold in main memory. For improving memory consumption, [95] proposes perfect hash-tables, with a hash function that assigns each state to a unique number. In our simpler setting, each state $S|_R = \bigcup_{i \in I} a_i$, for the index set $I$ and atom list $(a_i)_{i \in I}$, is hashed to $\sum_{i \in I} 2^i$, for a maximum of $2^{|R|}$ hash addresses.

## 10.3 Symbolic Pattern Databases

We abstract from the internal representation of sets of states as binary decision diagram (BDDs) [50]. It suffices to know that there BDDs are unique, space-efficient data structures for representing and manipulating Boolean formulae.

### 10.3.1 States and Operators

Boolean formulae may represent sets of states.

**Definition 8** *A* symbolic representation *for a state* $S \in \mathcal{S}$ *with* $\mathcal{S} \subseteq 2^A$ *is a set of boolean variables* $b_1, \ldots, b_{|A|}$, *with* $b_i$ *encoding the truth of propositional atom* $a_i$ *in a given state,* $i \in N = \{1, \ldots, |A|\}$.

*If* $S = \bigcup_{i \in I} a_i$, *then its encoding is* $(\bigwedge_{i \in I} b_i) \wedge (\bigwedge_{i \in N \setminus I} \neg b_i)$. *Sets of states* $\bigcup_{j \in J} S_j$ *are encoded as* $\bigvee_{j \in J} ((\bigwedge_{i \in I_j} b_{ij}) \wedge (\bigwedge_{i \in N \setminus I_j} \neg b_{ij}))$.

Transitions relations are Boolean expressions for operator application. They encode all valid (state, successor state) pairs utilizing twice the number of Boolean state variables; $2 \cdot |A|$ in our case. In practice, the transition relation is generated as the disjunct of the representations of all grounded operators, which in turn are defined as Boolean expressions of their precondition, add and delete lists.

**Definition 9** *The* transition relation $T(b, b')$ *of a set of operators* $O \in \mathcal{O}$ *is the disjunct* $T(b, b') = \bigvee_{O \in \mathcal{O}} T_O(b, b')$. *For* $O = (\alpha, (\beta_a, \beta_d))$ *we have* $T_O(b, b') = (\bigwedge_{a_i \in \alpha} b_i) \wedge (\bigwedge_{a_j \in \beta_a} b'_j) \wedge (\bigwedge_{a_k \in \beta_d} \neg b'_k) \wedge \text{frame}(b, b')$, *where* frame *encodes that all other atoms are preserved, i.e.* $\text{frame}(b, b') = \bigwedge_{a_j \notin \alpha \cup \beta_a \cup \beta_b} (b_j \equiv b'_j)$.

*Similarly, the relaxed* transition relation $T|_R$ *according the set of proposition* $R$ *is constructed with respect to the set of operators* $O|_R = (\alpha|_R, (\beta_a|_R, \beta_d|_R))$

The image $I$ of the state set *From* with respect to the transition relation $T$ is computed as $I(b') = \exists b' \; (T(b, b') \;\wedge\; \mathit{From}(b'))$. In this image computation, $T(b, b')$ is not required to be built explicitly, since with $T(b, b') = \bigvee_{O \in \mathcal{O}} T_O(b, b')$ we have $I(b') = \bigvee_{O \in \mathcal{O}} (\exists b' \; T_O(b, b') \;\wedge\; \mathit{From}(b'))$.

Therefore, the monolithic construction of $T(b, b')$ can be bypassed. Our current implementation organizes the image computation in form of a balanced tree. Through the success of conjunctive partitioning and reordering techniques in hardware verification [263], refined disjunctive partitioning approaches are an apparent issue for future research.

### 10.3.2 Pattern Database Construction

Complete symbolic breadth-first search (BFS) is one form of reachability analysis of the planning space. Let $\mathcal{S}_i$ be the set of planning states reachable from the initial state $S$ in $i$ steps, initialized by $\mathcal{S}_0 = \mathcal{I}$. An encoded state $S$ belongs to $\mathcal{S}_i$ if it has a predecessor $S'$ in the set $\mathcal{S}_{i-1}$ and there exists an operator which transforms $S'$ into $S$. All sets of states are identified by their respective characteristic formulae.

We apply backward symbolic exploration for SPDB construction as follows. The symbolic PDB $\mathcal{PDB}|_R$, is initialized to $\mathcal{G}|_R$ and, as long as there are newly encountered states, we take the current list of horizon nodes and generate the predecessor list with respect to $T|_R$. Then we attach the current BFS level to the new states, merge them with the set of already reached state states and iterate. In the following algorithm *Construct Symbolic Pattern Database*, *Reached* is the set of visited states, *Open* is current search horizon, and *Pred* is the set of predecessor states.

**Algorithm** *Construct Symbolic Pattern Database*
**Input:** *Planning space abstraction*
$\mathcal{P}|_R = < \mathcal{S}|_R, \mathcal{O}|_R, \mathcal{I}|_R, \mathcal{G}|_R >$
**Output:** *Symbolic Pattern Database* $\mathcal{PDB}|_R$
$\mathit{Reached}(b') \leftarrow \mathit{Open}(b') \leftarrow \mathcal{G}|_R(b')$
$i \leftarrow 0$
**while** ($\mathit{Open} \not\equiv \emptyset$)
$\mathit{Pred}(b) \leftarrow \exists b' \; \mathit{Open}(b') \wedge T|_R(b, b')$
$\mathit{Pred}(b') \leftarrow \mathit{Pred}(b) \; [b \leftrightarrow b']$
$\mathit{Open}(b') \leftarrow \mathit{Pred}(b') \wedge \neg \; \mathit{Reached}(b')$

$\mathcal{PDB}|_R \leftarrow \mathcal{PDB}|_R \vee (v = i \wedge \textit{Open}(b'))$
$\textit{Reached}(b) \leftarrow \textit{Reached}(b) \vee \textit{Open}(b)$
$i \leftarrow i + 1$
**return** $\mathcal{PDB}|_R$

Weightening the heursitic estimate according to a factor $\gamma$ is achieved by setting $(v = i)$ to $(v = \gamma i)$.

Note that beside the capability to represent large sets of states in the exploration, symbolic PDB have one further advantage to explicit ones: fast initialization. In the definition of most planning problems $\mathcal{G}$ is not given as a collection of states, but as a smaller selection of atoms $a_i$, $i \in I' \subset I$. In explicit PDB construction all states $G \in \mathcal{G}$ have to be generated and to be inserted into the BFS queue, while for the symbolic construction, initialization is immediate.

## 10.4 Explicit Pattern Database Search

SPDBs can easily be incorporated to any explicit heuristic search engine, e.g. Algorithm *Explicit Pattern Database Search* illustrates A* exploration with SPDBs.

**Algorithm** *Explicit Pattern Database Search*
**Input:** *Planning space* $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$,
*Symbolic Pattern Database* $\mathcal{PDB}|_R$
**Output:** Solution length $\delta(\mathcal{I})$
$\textit{Insert}(\textit{Open}, (\mathcal{I}, \mathcal{PDB}|_R(\mathcal{I})))$
**while** $(\textit{Open} \neq \emptyset)$
$S \leftarrow \textit{DeleteMin}(\textit{Open})$
**if** $(S \in \mathcal{G})$ **return** $g(S)$
**for all successors** $S'$ **of** $S$
$f(S') \leftarrow f(S) + 1 + \mathcal{PDB}|_R(S') - \mathcal{PDB}|_R(S)$
**if** $(\textit{Search}(\textit{Open}, S))$
**if** $(f'(S) < f(S))$
$\textit{DecreaseKey}(\textit{Open}, (S, f'(S))$
**else** $\textit{Insert}(\textit{Open}, (S, f'(S))$

The set of horizon nodes *Open* is represented as a priority queue with usual access operations *DeleteMin*, *Insert*, and *DecreaseKey*. For the sake of brevity, we have omitted re-opening and concentrate on only one PDB, since generalizations to planning pattern partitions $\mathcal{PDB}|_Q$ are easy to obtain.

For each extracted state $S$ we have $f(S) = g(S) + h(S)$, where $g$ is the actual distance to state $S$. The new $f$-value of a successor $S'$ is calculated as $f(S') = g(S') + h(S') = g(S) + 1 + h(S') = f(S) + 1 + (h(S') - h(S))$.

Apparently, the design of explicit search algorithms with symbolic PDB heuristics is not different to the design of algorithms for any other incorporated estimate. The only change is the computation of the estimate $h(S)$ for state $S$ with respect to $\mathcal{PDB}|_R$.

To query the symbolic PDB $\mathcal{PDB}|_R$ with state $S = \bigcup_i a_i$, denoted as $\mathcal{PDB}|_R(S)$, we first compute the symbolic representation $\bigwedge_i b_i$ of $S$. Then we determine the conjunct

of $\bigwedge_i b_i$ with $\mathcal{PDB}|_R$. The operation yields $(\bigwedge_j v_j) \wedge \ (\bigwedge_i b_i)$, where $(\bigwedge_j v_j)$ encodes the estimate. Last but not least, the formula $(\bigwedge_j v_j)$ is converted back to an ordinary numerical quantity. Since $\bigwedge_i b_i$ is already simple, computing its conjunct with $\mathcal{PDB}|_R$ is fast in practice. Conversion would not be necessary at all, if instead of BDDs – as in our implementation – arithmetic decision diagrams (ADDs) were used. For this case, the heuristic estimate is determined in time linear to the encoding length. If several SPDs $Q_1, \ldots, Q_k$ are addressed, we compute the estimate $h(S) = h_1(S) + \ldots + h_k(S)$ with respect to $h_1(S) = \mathcal{PDB}|_{Q_1}(S)$, $h_2(S) = \mathcal{PDB}|_{Q_2}(S)$, ..., $h_k(S) = \mathcal{PDB}|_{Q_k}(S)$.

## 10.5 Symbolic Pattern Database Search

In the symbolic version of heuristic search the algorithm determines all successor states for a set of successors in one evaluation step. The heuristic is represented as a binary relation of estimate and state variables. In the exploration algorithm the open list of generated nodes is represented as an encoded set of buckets with bucket $f$ containing all states in open with merit $f = g + h$ .

Algorithm *Explicit Pattern Database Search* starts with the Boolean representation of the initial state, attaches its estimate and similarly to the explicit case, it iterates state extraction and successor set generation until the goal has been found. However, in contrast to the setting above, we extract sets of states *Min* with minimum $f$-value $f_{\min}$ and compute their respective successor sets *Succ* by applying the transition relation. To find the $f$-value for the successor states we apply symbolic representation of the heuristic estimator $\mathcal{PDB}|_R$ to the pre-image and the image of transition relation application. The correctly associated values $h, h'$ are then quantified to yield the successor $f$-value ($f = f_{\min} + h' - h + 1$). For best-first search the formula simplies to $f = h'$.

The superimposed distribution $\mathcal{PDB}|_{R+Q}$ of two PDBs $\mathcal{PDB}|_R$ and $\mathcal{PDB}|_Q$ approximates $\mathcal{PDB}|_{R \cup Q}$. It can be computed beforehand to be conjuncted with *Min* and *Succ* in the algorithm. The alternative avoids the pre-computation of $\mathcal{PDB}|_{R+Q}$ and combines $\mathcal{PDB}|_R$ and $\mathcal{PDB}|_Q$ with *Min* and *Succ* during the execution. Our implementation allows both options.

Given a uniform weighted problem graph and a consistent heuristic ($h(v) - h(u) + w(u,v) \geq 0$) the worst-case number of iterations has been shown to be $O(\delta^2(\mathcal{I}))$ [112].

**Algorithm** *Symbolic Pattern Database Search*
**Input:** *Planning space* $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$,
*Symbolic Pattern Database* $\mathcal{PDB}|_R$
**Output:** Solution length $\delta(\mathcal{I})$
$Open(f,b) \leftarrow \mathcal{PDB}|_R(f,b) \wedge \mathcal{I}(b)$
**do**
$f_{\min} = \min\{f \mid f \wedge Open(f,b) \neq \emptyset\}$
$Min(f) \leftarrow \exists f \ (Open(f,b) \wedge f = f_{\min})$
$Rest(f,b) \leftarrow Open(f,b) \ \wedge \ \neg \ Min(b)$
$Min(h,b) \leftarrow \mathcal{PDB}|_R(h,b) \wedge Min(b)$
$Succ(h,b') \leftarrow \exists b \, T(b,b') \wedge Min(h,b)$
$Succ(h,b) \leftarrow Succ(h,b') \ [b \leftrightarrow b']$
$Succ(h,h',b) \leftarrow \mathcal{PDB}|_R(h',b) \wedge Succ(h,b)$

$Succ(h, h', f, b) \leftarrow Succ(h, h', b) \; \wedge \; f = f_{\min}$
$Succ(f', b) \leftarrow \exists h, h', f$
$\quad Succ(h, h', f, b) \wedge (f' = f + h' - h + 1)$
$Succ(f, b) \leftarrow Succ(f', b) \; [f \leftrightarrow f']$
$Succ(f, b) \leftarrow Succ(f, b) \wedge \neg Reached(b)$
$Open(f, b) \leftarrow Rest(f, b) \vee Succ(f, b)$
$Succ(b) \leftarrow \exists f \; Succ(f, b)$
$Reached \leftarrow Reached(b) \vee Succ(b)$
**while** ($Open \wedge \mathcal{G} \equiv \emptyset$)
**return** $f_{\min}$

## 10.6 Search Tree Prediction

Heuristic PDBs are also an efficient mean for heuristic search tree prediction, since they approximate the overall distribution of heuristic estimates in the state space. Assuming that patterns occur equally likely in the search space, the overall probability of estimate $h$ being less than or equal to $k$ is

$$P(h \leq k) = |\{P \in \mathcal{PDB} \mid h(P) \leq k\}| / |\mathcal{PDB}|.$$

To predict the heuristic search tree expansion of the problem graph that is labeled with node costs $f = g + h$, the main result in [220] states that the expected total number of tree nodes according to cost threshold $\theta$ is approximately equal to

$$\sum_{d=0}^{\theta} n^{(d)} P\left(h \leq \theta - d\right), \tag{10.1}$$

where $n^{(d)}$ is the number of states in the brute-force search tree with depth $d$ and $P$ is the equilibrium distribution, defined as the probability distribution of heuristic values in the limit of large depth. In the framework of spectral analysis, $n^{(d)}$ can be computed in closed form [96].

Since Equation 10.1 is a very good predictor for the number of nodes in IDA* and yields at least a good trend for A*'s exploration effort, it has been used for evaluating the effectiveness of PDBs [173]. For the sake of simplicity we focus on the mean heuristic value $\overline{h} = \sum_k k \cdot |\{P \in \mathcal{PDB} \mid h(P) = k\}| / |\mathcal{PDB}|$. In the limit of large $\theta$, the branching factor $b$, i.e. the ratio of search tree nodes with respect to two consecutive threshold values, converges [96]. The effect of the heuristic is to reduce the search tree size from $O(b^d)$ to $O(b^{d-c})$ for some constant $c \approx \overline{h}$ [215].

Therefore, heuristics are best thought of as offsets to the search depth. The higher the average heuristic value, the smaller the *effective search tree depth*, i.e., the shallower the search with respect to the brute-force search tree. The following case study displays the effect of explicit and symbolic PDBs on $\overline{h}$.

## 10.7 Case Study

As a case study we chose Blocks World, since finding optimal plans is still a challenge for domain-independent planners. No form of knowledge was added to the planner, we

switched off all branching cuts. Cuts significantly speed up exploration, but most proposed control knowledge in planning is domain-dependent or apply to certain sets of benchmark domains only.

### 10.7.1 Pattern Database Construction

In Tables 10.1 and 10.2 we present the results on constructing symbolic PDBs in selected Blocks World problems of the AIPS-2000 set. The total number of pattern $s$ in the databases and the respective averaged heuristic estimates $\overline{h}$ are shown. As the problems size $p$ gets larger, more and more PDBs were generated (separated by /).

| $p$ | $s$ | $\overline{h}$ |
|---|---|---|
| 4 | 108 | 6.20 |
| 5 | 1,029 | 8.85 |
| 6 | 12,288 | 11.49 |
| 7 | 26,244/8 | 11.72/1.62 |
| 8 | 50,000/80 | 11.26/3.57 |
| 9 | 87,846/968 | 11.69/6.37 |
| 10 | 145,152/13,824 | 11.35/8.59 |
| 11 | 228,488/228,488 | 11.38/11.39 |
| 12 | 27,440/27,440/2,156 | 8.64/8.64/5.79 |
| 13 | 37,125/37,125/37,125 | 8.66/8.66/8.66 |
| 14 | 49,152/49,152/49,152/15 | 8.03/8.03/8.03/1.80 |
| 15 | 63,869/63,869/63,869/255 | 8.69/8.69/9.35/3.75 |

Table 10.1: Number of States $s$ and Mean Heuristic Value $\overline{h}$ in Blocks World PDBs according to $m = 2^{20}$.

| $p$ | $s$ | $\overline{h}$ |
|---|---|---|
| 4 | 1,08 | 6.20 |
| 5 | 1,029 | 8.85 |
| 6 | 12,288 | 11.49 |
| 7 | 1,777,147 | 14.14 |
| 8 | 3e+06 | 16.80 |
| 9 | 5.84e+07 | 19.47 |
| 10 | 1.43e+08 | 19.55/1.72 |
| 11 | 2.89e+07/1,690 | 17.05/6.47 |
| 12 | 5.27e+07/27,440 | 16.29/8.05 |
| 13 | 9.11e+07/506,250 | 16.89/10.93 |
| 14 | 1.50e+08/1.04e+07 | 16.56/13.77 |
| 15 | 2.41e+08/2.41e+08 | 16.59/16.56 |

Table 10.2: Number of States $s$ and Mean Heuristic Value $\overline{h}$ in Blocks World PDBs according to $m = 2^{30}$.

The averaged heuristic estimates increase significantly when moving from $m = 2^{20}$ to $m = 2^{30}$, while the number of PDBs shrinks accordingly.

Table 10.3 compares the growth of the symbolic representation with respect to the number of states. We took the first PDB in Blocks World Problem 15 with $m = 2^{30}$ as an example. The predicted state space size is 410,338,673. Since this corresponds to maximum perfect hash table capacity, explicit exploration was no longer available.

| $d$ | $t$ | $b$ | $s$ |
|---|---|---|---|
| 0 | 0.00s | 35 | 1 |
| 2 | 0.01s | 103 | 39 |
| 4 | 0.01s | 311 | 586 |
| 6 | 0.03s | 858 | 5,792 |
| 8 | 0.13s | 2,576 | 55,911 |
| 10 | 0.66s | 6,879 | 538,771 |
| 12 | 2.56s | 14,583 | 4.01e+06 |
| 14 | 7.60s | 24,547 | 1.87e+07 |
| 16 | 12.60s | 30,238 | 4.51e+07 |
| 18 | 10.69s | 22,592 | 4.14e+07 |
| 20 | 4.48s | 7,655 | 1.02e+07 |
| 22 | 0.45s | 993 | 467,551 |

Table 10.3: Node Count $b$ and Number of States $s$ for Constructng a SPDB in Blocks-World Problem 15.

Table 10.3 depicts the node and state counts for each iteration in the construction phase. The results indicate that by far more states are encountered than BDD nodes were necessary to represent them. In this case the effect of symbolic representation corresponds memory gains of up to about two orders of magnitude. With $m = 2^{40}$, for which exploration was still possible, the effect increases up to about four orders of magnitude. We also observe that the peak node count for the is also established earlier then the peak state count.

### 10.7.2 Explicit Search

Table 10.4 compares the CPU times[1] of explicit and symbolic PDB construction with the exploration time in explicit search. We took the same heuristic estimate and $m = 2^{20}$. Since the qualities of the different PDBs match, the same set of states was considered. The search algorithm we chose was A* with weight 2 ($f = g + 2h$). Besides the problem number, the depth of the solution and the number of expanded nodes, we also displayed PDB construction time $t^c$ and explicit search time $t^s$ with respect to explicit pattern and symbolic PDBs, subscripted by $e$ and $s$.

[1]Most of the experiments were run on a Sun UltraSparc Workstation with 248 MHz. Since exact running-times reflect too many issues of the current implementation, for the interpretation of results we are mainly interested in comparing performance growth. Memory was restricted as follows. The pattern databases were either limited to $m = 2^{20}$ states or $m = 2^{30}$ states; for explicit search we chose 2,000,000 stored states as the exploration bound. For symbolic exploration we allocated at most 8,000,000 BDD

| $p$ | $d$ | $e$ | $t_e^c$ | $t_e^s$ | $t_s^c$ | $t_s^s$ |
|---|---|---|---|---|---|---|
| 4 | 6 | 7 | 0.01s | 0.00s | 0.03s | 0.00s |
| 5 | 12 | 15 | 0.05s | 0.00s | 0.04s | 0.00s |
| 6 | 12 | 13 | 0.49s | 0.00s | 0.30s | 0.00s |
| 7 | 24 | 40 | 1.39s | 0.00s | 0.52s | 0.01s |
| 8 | 20 | 1,590 | 2.98s | 0.10s | 0.67s | 0.40s |
| 9 | 32 | 34,316 | 6.18s | 3.75s | 0.81s | 13.92s |
| 10 | 34 | 47,657 | 12.55s | 5.72s | 1.23s | 16.35s |
| 11 | 38 | 7,941 | 0.91s | 3.09s | 1.80s | 3.04s |
| 12 | 38 | 34,323 | 4.67s | 5.31s | 1.73s | 13.24s |
| 13 | - | - | 10.55s | - | 2.45s | - |
| 14 | 40 | 254,769 | 15.16s | 58.23s | 3.32s | 150.43s |
| 15 | - | - | 21.88s | - | 5.51s | - |

Table 10.4: Time for PDB Construction and Explicit Search in Blocks World.

In the result we obtain a trade-off between explicit and symbolic search. While symbolic PDB construction is significantly faster, search time is larger. As indicated above, an ADD implementation for the heuristic lessens the per-node retrieval overhead for SPDBs.

### 10.7.3 Symbolic Search

In this set of experiments we measured the performance of the symbolic search algorithm. We used forward heuristic search with respect to the provided SPDBs, accompanied by a symbolic backward traversal. The search direction was chosen in favor to the exploration side that used less time in the previous iteration. The memory bound was set to $m = 2^{30}$, so at most 2 PDBs were constructed. By the choice of dependent PDBs, the results in Table 10.5 were not necessarily optimal. The headings are read as follows: $p$ is the problem number, $d$ is the depth of the solution, $i_f$ and $i_b$ are the number of forward and backward iterations, $t_s^c$ is the PDB construction time, $t_s^c$ is the symbolic search time, and $t_b$ is time for bidirectional symbolic BFS.

The peak PDBs size at $p = 11$ reflects that the maximum number of patterns in the database is roughly equal to the state space size. As the comparison of $t_s^s$ with $t_b$ shows, we can obtain better results as with bi-directional symbolic BFS, which besides SAT enumeration [204], is state-of-the-art in optimal sequential plan generation. Another observation is that in case of failure, symbolic heuristic search with PDBs never runs out of memory but out of time. For symbolic engines this is a very unusual behavior. In Problem 13 time was exceeded in exploration, while for Problem 15 the time threshold was encountered when merging the two PDBs into a combined one.

PDBs have also been successfully applied to other challenging propositional planning domains [95]. The results do transfer to the symbolic setting. In simple domains like Gripper, all running times (for A* and best-first explicit and symbolic exploration with explicit and symbolic PDBs) were bounded by far less than a minute. Table 10.6 displays the CPU performance of explicit search with (S)PDBs in Logistics. In symbolic

---

nodes.

| $p$ | $d$ | $i_f$ | $i_b$ | $t_s^c$ | $t_s^s$ | $t_b$ |
|---|---|---|---|---|---|---|
| 4 | 6 | 6 | 0 | 0.02s | 0.21s | 0.17s |
| 5 | 12 | 12 | 0 | 0.04s | 0.30s | 0.30s |
| 6 | 12 | 12 | 0 | 0.30s | 0.43s | 1.09s |
| 7 | 20 | 20 | 0 | 3.95s | 0.76s | 11.34s |
| 8 | 18 | 9 | 11 | 0.67s | 0.40s | 2.80s |
| 9 | 30/32 | 25 | 12 | 0.81s | 13.92s | 38.16s |
| 10 | 34 | 60 | 12 | 66.16s | 58.02s | 297.51s |
| 11 | 32/38 | 52 | 11 | 1,218s | 261.76s | 742.14s |
| 12 | 34/36 | 142 | 15 | 38.57s | 224.13s | 1,059s |
| 13 | - | 147 | 17 | 48.88s | time | memory |
| 14 | -/38 | 52 | 11 | 59.05s | 150.92s | memory |
| 15 | - | - | - | time | - | memory |

Table 10.5: CPU Performance for PDB Construction and Symbolic Search in Blocks World.

best-first search ($f = h$) we solved each problem in less than a minute, while symbolic A* ($f = g + h$) and symbolic BFS ($f = g$) failed in larger problem instances. The space bound is $2^{20}$ and, once more, the search algorithm is A* with weight 2. The savings

| $p$ | $d$ | $e$ | $t_e^c$ | $t_e^s$ | $t_s^c$ | $t_s^s$ |
|---|---|---|---|---|---|---|
| 4 | 20 | 21 | 1.52s | 0.00s | 0.11s | 0.00s |
| 5 | 27 | 33 | 0.70s | 0.01s | 0.08s | 0.02s |
| 6 | 25 | 30 | 5.90s | 0.00s | 0.10s | 0.07s |
| 7 | 37 | 48 | 27.54s | 0.01s | 0.64s | 0.06s |
| 8 | 34 | 50 | 26.99s | 0.02s | 2.33s | 0.06s |
| 9 | 36 | 43 | 27.62s | 0.01s | 0.91s | 0.08s |
| 10 | 36 | 81 | 52.97s | 0.04s | 1.01s | 0.12s |
| 12 | 44 | 79 | 53.02s | 0.02s | 1.14s | 0.11s |
| 13 | 75 | 138 | 22.92s | 0.08s | 4.18s | 0.42s |
| 14 | 66 | 143 | 22.99s | 0.09s | 4.42s | 0.36s |
| 15 | 84 | 186 | 23.39s | 0.15s | 4.81s | 0.51s |

Table 10.6: Time for PDB Construction and Explicit Search in Logistics.

in explicit database search time are counter-balanced by acorresponding increase in construction time. One interesting observation in Logistics and Gripper is that through the highly asynchronous problem structure even very small databases lead to good accumulated estimates. Therefore, very large problems can effectively be solved with PDB.

We have not considered metric planning problems, where PDBs are to be constructed according to their shortest-path distances to the goal. Since the awarded, 2002 competition version of MIPS [2] schedules sequential plans, we integrated (S)PDB for sequential plan generation with mixed results.

[2]See `www.informatik.uni-freiburg.de/~mmips`.

## 10.8 Related Work

In the Model-Based Planner, MBP, the paradigm of planning as symbolic model checking [145] has been implemented for *non-deterministic planning* domains [63], which classifies in weak, strong, and strong-cyclic planning, with plans that are represented as complete state-action tables. For *partial observable planning*, exploration faces the space of belief states; the power set of the original planning space. Therefore, in contrast to the successor set generation based on action application, observations introduce “And” nodes into the search tree [32]. Since the approach is a hybrid of symbolic representation of belief states and explicit search within the “And”-“Or” search tree, simple heuristic have been applied to guide the search. The need for heuristics that trade information gain for exploration effort is also apparent need in *conformant planning* [31]. The authors label the obtained search algorithms as a new paradigm of *heuristic-symbolic* search and report savings in orders of magnitudes with respect to BFS. In contrast to our approach, where Boolean function encode perfect knowledge, the symbolic representation compensates partial knowledge of the current state. Moreover, Bertoli et al. consider heuristics for guiding the choice of the belief states with no symbolic heuristic estimates as in our case. Since the first estimate was rather trivial – it preferred belief states with low cardinality – recent work [30] proposes improved heuristic for belief space planning. Nevertheless, we view unpublished work on abstraction [61] closest to our approach of symbolic PDBs. It origins in Abstrips abstractions, but lacks experimental results.

The awarded model checking integrated planning system MIPS [103] is a competitive deterministic planning system based on model checking methods. The planner incorporates symbolic, explicit and metric heuristic planning strategies [100]. Its type-inference mechanism and fact enumeration algorithm groups mutually exclusive facts to infer a concise state endoding [101]. Heuristic symbolic search with the (weighted) BDDA* algorithm has shown a significant time and space reduction for planning problems that were intractable for breadth-first symbolic exploration [93]. As a symbolic heuristic, the goal was splitted into atoms and either a relaxed plan or the *single-atom* heuristic was computed and accumulated. The approach could not compete with state-of-the art planners, and, different to SPDBs, the pre-compiled symbolic estimates provided no information gain to accelerate explicit heuristic search planners.

The UMOP system parses a non-deterministic agent domain language that explicitly defines a controllable system in an uncontrollable environment [196]. The planner also applies BDD refinement techniques such as automated transition function partitioning. New result for the UMOP system extends the setting of weak, strong and strong cyclic planning to adversarial planning, in which the environment actively influences the outcome of actions. In fact, the proposed algorithm joins aspects of both symbolic search and game playing. Jensen also reports some preliminary and unpublished successes on planning with domain abstractions. As one drawback, the loss of solution quality seemed to be significant.

With SetA*, [197] provide an improved implementation of the symbolic heuristic search algorithm BDDA* [112] and Weighted BDDA* [93]. Based on supplied source code the consise state encoding and the *max-atom* heuristic function of MIPS could be reproduced[3]. One major surplus is to maintain a finer granularity of the sets of states in

[3]In their paper, the authors compare SetA* with the implementation of BDDA* in MIPS of early 2001. While the results in Logistics seem plausible, unfortunately, we cannot reproduce the bad behavior of our

the search horizon kept in a matrix according to matching $g$- and $h$- values. This contrasts the plain bucket representation of the priority queue based on $f$-values. The heuristic function is implicitly encoded with value differences of grounded actions. Since sets of states are to be evaluated and some heuristics are state rather than operator dependent it has still to be shown how general this approch is. As above the considered planning benchmarks are seemingly simple for single-state heuristic search exploration [180, 169]. We expect better and more general results when applying SPDBs.

Recent, yet unpublished work of Hansen, Zhou, and Feng [158] also re-implemented BDDA* and suggest that symbolic search heuristics and exploration schemes are probably better to be implemented with algebraic decision diagrams (ADDs), as available in Somenzi's CUDD package. Although the authors achieved no improvement to [112] to solve the $(n^2-1)$-Puzzle, the established generalization to guide a symbolic version of the LAO* exploration algorithm [157] for *probabilistic* or Markov decision process (MDP) planning, results in a remarkable improvement to the state-of-the-art [124]. Since its input – as in our case – is a symbolic representation of the estimate, the contributed progress in estimate quality calls for generalizations of SPDBs to probabilistic planning.

In BDD-based hardware verification, guided search and prioritized model checking are emerging technologies. [352] used BDD-based symbolic search based on the Hamming distance of two states. This approach has been improved in [303], where the BDD-based version of A* for the $\mu$cke model checker outperforms symbolic BFS exploration for two scalable hardware circuits. The heuristic is determined in a static analysis prior to the search taking the actual circuit layout and the failure formula into account. The approach of symbolic guided search in CTL model checking documented in [38] applies `hints' to avoid sections of the search space that are difficult to represent for BDDs. This permits splitting the fix-point iteration process used in symbolic exploration into two parts yielding under- and over-approximation of the transition relation. Benefits of this approach are simplification of the transition relation, avoidance of BDD blowup and a reduced amount of exploration for complicated systems. However, in contrast to our approach providing `hints' requires user intervention. Also, this approach is not directly applicable to explicit exploration, which is our main focus. Prioritized traversals are also concerned for formal hardware verification at IBM [133]. The approach bases on the work of [54] and splits the symbolic search frontier into parts to ease approximate reachability.

## 10.9 Conclusion

This paper puts forth the idea of PDB construction to improve the computed average of the admissible heuristic, which in turn corresponds to a relative decrease in search depth. We have also seen a sound formal treatment for PDBs in planning for both explicit and symbolic construction. The experiments highlight that with symbolic representation and reachability analysis, very large databases can be constructed, for which explicit methods necessarily fail.

The approach improves one of the three major classes of heuristics in planning, namely *Plan abstraction*. The other two are: *Plan relaxation*, as implemented in the FF planner [181], which is a informative on-line computed estimate,

implementation in the Gripper domain.

and *Bellman approximation*, as implemented in the *max-atom* and *max-pair* heuristics for HSP, which also consideres groups of atoms. In contrast to this paper *Bellman approximation* simplifies the exploration without simplifying the operator representation [163].

PDBs consider subproblem interactions of larger groups and include more knowledge into the estimate than the *max-pair* heuristic. On the other hand, since FF and the PDB heuristics are very different in their characteristics, the natural question arises of how to combine the two for an even better estimate. Even though node expansion is more time consuming for the relaxed plan graph estimate, it yields better information on groups that do not appear in the goal description.

Our implemented proposal is to group the number of add atoms that match the backward plan extraction of the relaxed plan graph in FF according to the obtained group partitioning. With respect to each planning space abstraction the better FF or PDB, value can be selected. Since FF's heuristic is somewhat misguided in Blocks World, yielding very low estimates in states far away from the goal state, we can achieve almost arbitrary large improvements for A*-like searches.

Our approach accelerates both explicit and symbolic search. Explicit heuristic search planners can now access better off-line estimates and by weighting the symbolic heuristic search algorithm we can scale the solution quality. Symbolic heuristic search planning – possibly better to be implemented with ADDs – now appears as a real competitor for *blind* symbolic breadth-first exploration. Moreover, the paper provides a bridge from explicit to symbolic search. In both planning and model checking there are two distinctive research branches according to the chosen representation. We have established an effective interplay between these methods by combining state-of-the art techniques from both fields. Future research on checking safety property will try to consolidate these findings in model checking domains.

# Paper 11

# Taming Numbers and Durations in the Model Checking Integrated Planning System

Stefan Edelkamp.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg

In *Journal of Artificial Research*, 2003. To appear.

**Abstract**

The Model Checking Integrated Planning System (MIPS) has shown distinguished performance in the second and third international planning competitions. With its object-oriented framework architecture MIPS clearly separates the portfolio of explicit and symbolic heuristic search exploration algorithms from different on-line and off-line computed estimates and from the grounded planning problem representation.

In 2002, the domain description language for the benchmark problems has been extended from pure propositional planning to include rational state resources, action durations, and plan quality objective functions. MIPS has been the only system that produced plans in each track of every benchmark domain. This article presents and analyzes the algorithmic novelties necessary to tackle the new layers of expressiveness.

The planner extensions include critical path analysis of sequentially generated plans to generate optimal parallel plans. The linear time algorithm bypasses known NP hardness results for partial ordering with mutual exclusion by scheduling plans with respect to the set of actions *and* the imposed causal structure. To improve exploration guidance approximate plans are scheduled for each encountered planning state.

One major strength of MIPS is its static analysis phase that grounds and simplifies parameterized predicates, functions and operators, that infers single-valued invariances to minimize the state description length, and that detects symmetries of domain objects. The aspect of object symmetry is analyzed in detail. The paper shows how temporal plans of any planner can be visualized in Gannt-chart format in a client-server architecture. The frontend turns also be appropriate for concise domain visualization.

## 11.1 Introduction

The Model Checking Integrated Planning System MIPS has participated twice in the international planning competition: in the second planning competition at AIPS-2000 in Beckenridge (USA) and in the third planning competition at AIPS-2002 in Toulouse (France). As the name indicates, the MIPS project targets the integration of model checking techniques into a domain-independent action planner.

*Model checking* [65] is the automated process to verify if a formal model of a system satisfies an specified temporal property or not. As an illustrative example, take an elevator control system together with a correctness property that requires an elevator to eventually stop on every call of a passenger or that guarantees that the door is closed, while the elevator is moving.

Although the success in checking correctness is limited, model checkers found many subtle errors in current hardware and software designs. Models often consists of many concurrent sub-systems. Their combination is either synchronous, as often met in hardware design verification, or asynchronous, as frequently given in communication and security protocols, or in multi-threaded programming languages like Java.

Exploration of model checking domains spans very large spaces of all reachable system states. This effect is usually denoted as the *state explosion problem*, even if the sets of generated states rather than the states themselves grow that quickly.

An error that shows a safety property violation, like a deadlock or a failed assertion, corresponds to one of a set of target nodes in the state space graph. Roughly speaking, *something bad has occured*. A liveness property violation refers to a (seeded) cycle in the graph. Roughly speaking, *something good will never occur*. For the case of the elevator example, eventually reaching a target state where a request button was pressed is a liveness property, while certifying closed doors refers to a safety property.

In this paper we refer to safety properties only, since goal achievement in traditional and competition planning problems have yet not been extended with temporal properties. However, temporally extended goals are of increasing research interests [200, 294, 230].

The two main validation processes in model checking are explicit and symbolic search. In explicit-state model checking each state refers to a fixed memory location and the state space graph is implicitly generated by successive expansions of state.

In symbolic model checking [259, 66], fixed-length binary encodings of states are usually seen as mandatory, so that each state can be represented by its characteristic Boolean function. The function evaluates to true if and only if all state variables are assigned to according bit values. Sets of states are expressed by the disjunct of the individual characteristic functions. On the other hand satisfiability and uniqueness of Boolean formulae is NP hard.

The unique symbolic representation of sets of states as Boolean formulae through binary decision diagrams (BDDs) [50] is often much smaller than the explicit one. BDDs are (ordered) read-once branching programs with nodes corresponding to variables, edges corresponding to variable outcomes, and each path corresponding to an assignment to the variables with the resulting evaluation at the leaves. One reason of the succinctness of BDDs is that directed acyclic graphs may express exponentially many paths. Since states are encoded in binary, the transition relation is defined on two state variable sets. It evaluates to true, if and only if an operator exists that transforms a state into a valid successor. In some sense, BDDs exploit regularities of the state set and often appear

better suited to regular hardware systems, in contrast to many software system that inherit a highly asynchronous and irregular structure, so that the straight use BDD with their fixed variable ordering is probably not flexible enough.

For symbolic exploration a set of states is combined with the transition relation to compute the set of all possible successor states, i.e. the image. Starting with the initial state, iteration of image computations eventually explores the entire reachable state space. To improve the efficiency of image computations, transition relations are often provided in partitioned form.

The correspondence of action planning and model checking can be roughly characterized as follows. Similar to model checkers, action planners implicitly generate large state spaces, and both exploration approaches base on applying parameterized operators to the current state. States in model checking and in planning problems are both labeled by (propositional state) predicates. The satisfaction of a specified property on the one side and the arrival at a certain goal state on the other, leads to a slight difference in the according search objective. With this respect, the goal in action planning is a safety error and the corresponding (error) trail is interpreted as a plan. In the elevator example, the goal of a planning task is to reach a state, in which the doors are open and the elevator is moving. For a formal treatment on the embedding of planning problems into model checking terminology, we refer the reader to [145].

Model checkers perform either symbolic or explicit exploration. To the contrary MIPS features both and allows to combines symbolic and explicit search planning. It applies heuristic search; a search acceleration technique that has let to considerable gains in both communities. In the last few years, heuristic search planners frequently outperform other domain-independent planning approaches, e.g. [181], and heuristic search model checkers turn out to significantly improve state-of-the-art, e.g. [105].

Including resource variables (like the fuel level of a vehicle or the distance between locations) and action duration (i.e. the time passed during execution of the planning operator) are relatively new aspects for action planning, at least in form of an accepted domain description accessible for competitive planning [131]. The competition input format PDDL2.1 is not restricted to variables of finite domain, but also includes specification of rational (floating-point) variables in both precondition and effects. Similar to a set of atoms described by a propositional predicate, a set of numerical quantities can be described by a set of parameters. Through the notation of PDDL2.1, we refer to parameterized numerical quantities as functions. For example, the fuel level might be parameterized by the vehicle that is present in the problem instance description.

In the 2002 competition, domains were provided in different tracks according to different layers of language expressiveness: $i$) pure propositional planning, $ii$) planning with numerical resources, $iii$) planning with numerical resources and constant action duration, $iv$) planning with numerical resources and variable action duration, and, in some cases, $v$) more complex problems usually combining time and numbers in more interesting ways. MIPS competed as a fully automated system and performed remarkably well in all five tracks; it solved a high number of problems and was the only system that produced solutions in each track of every benchmark domain.

In this paper the main algorithmic aspects to *tame* rational numbers, objective functions, and action duration are described. The article is structured as follows. First, we recall the development of the MIPS system and assert its main contributions to the planning community. Then we address the object-oriented heuristic search framework architecture

of the system. Subsequently, we fix some terminology that allows to give a formal definition of the syntax and the semantics of a grounded mixed numerical and propositional planning problem instance.

We then introduce the core contributions: critical path scheduling for concurrent plans, and efficient methods for detecting and using symmetry cuts. PERT scheduling produces optimal parallel plans given a sequence of operators and a precedence relation among them in linear time. The paper discusses pruning anomalies and handling of different optimization criteria. We analyze the correctness and efficiency of symmetry detection in detail. Afterwards, a TCP/IP client-server visualization system for sequential and temporal plans is presented. The article closes with related work and concluding remarks.

## 11.2 The Development of MIPS

The competing versions of MIPS refer to initial findings [114] of heuristic symbolic exploration with the $\mu$cke model checker [35] that already lead to good performance in puzzle solving [112] and in hardware verification [303]. For general propositional planning, our concise BDD library *StaticBdd*[1] has been used. During the implementation process we changed the BDD representation to improve performance mainly for small planning examples and chose the public domain `c++` BDD package Buddy by Jørn Lind-Nielsen. In the beginning the variable encodings were provided by hand, while the representation of all possible operator descriptions were established by enumerating all possible parameter instantiations. Once the encoding and transition relation were fixed, symbolic exploration in form of a reachability analysis of the state-space could be executed. At that time, we were not aware of any other work in BDD-based planning like [60], which is probably the first link to planning via (symbolic) model checking.

Since the above approach was criticized not to be fully automated, we subsequently developed a parser and static analyzer to cluster atoms into groups in order to minimize the length of the state encoding [101]. The outcome of the analyzer allowed to specify states and transition functions in Boolean terms, which in turn were included into a bidirectional BDD exploration and solution extraction procedure. In the end, MIPS was the first automated planning system based on symbolic model checking.

In the second international planning competition MIPS [103] could handle the STRIPS [126] subset of the PDDL language [258] and some additional features from ADL [289], namely negative preconditions and (universal) conditional effects. MIPS was one of five planning systems to be awarded for “Distinguished Performance” in the fully automated track. The competition version [102] already included explicit heuristic search algorithms based on a bit-vector state representation and the relaxed planning heuristic (RPH) [181] and symbolic heuristic search based on the HSP-Heuristic [41] and a one-to-one atom RPH-derivate [102].

For the 2002's international planning competition new levels of the planning domain description language [131] have been designed to specify problems that include actions with durations and resources. The agreed input language definition is referred to as PDDL 2.1. While Level 1 considers pure propositional planning, Level 2 also includes numerical resources and objective functions to be minimized, and Level 3 additionally allows to

[1] See `http://www.informatik.uni-freiburg.de/~edelkamp/StaticBdd`

specify actions with durations. Consequently, MIPS[2] has been extended to cope with these new forms of expressiveness.

In [94] first results of MIPS in planning PDDL 2.1 problems are presented. The preliminary treatment exemplifies the parsing process in two simple benchmark domains. Moreover, propositional heuristics and manual branching cuts were applied to accelerate sequential plan generation. This work was extended in [97], where two approximate exploration techniques to bound and to fix numerical domains, symmetry detection based on fact groups, critical path scheduling, an any-time wrapper to produce optimal plans and a numerical extension to the RPH were presented. Due to possible involved calculations, enumerating variable domains and the any-time wrapper were excluded from the competition version of MIPS. Our approach to extend RPH with numerical information establishes plans even in challenging numerical domains like *Settlers* and was developed independently from Hoffmann's work on his competing planner *Metric-FF* [179]. The main contributions of this paper are

- a formal definition of grounded propositional and numerical planning and an index scheme for grounding predicate, functions, and actions;
- the object-oriented framework architecture for a planner to choose and combine different heuristics with different search algorithms and storage structures;
- an intermediate interface with grounded and simplified planning domain instances;
- a static analyzer that applies efficient fact-space exploration to distinguish constant from variable atoms and resource variables, that clusters facts into groups and infers static symmetries;
- optimal temporal planning enumeration algorithms based on an precedence rala- tion and PERT scheduling of sequentially generated plans together with an concise analysis of correctness and optimality;
- the integration of scheduling already in the estimate computation as a floating point estimate for optimal parallel plan length;
- different pruning methods, especially dynamic symmetry detection, hash and transposition cuts, and a throughout study of object symmetries, their complexities and the implemented trade-off;
- different strategies for optimizing objective functions and further implementation tricks that made the system efficient;
- a web-interface for visualization: a wrapper for temporal plans to be presented in Gannt-chart format and a domain-dependent frontend for executing sequential plans.

[2] A recent version of MIPS is available in source code at `www.informatik.uni-freiburg.de/~edelkamp`

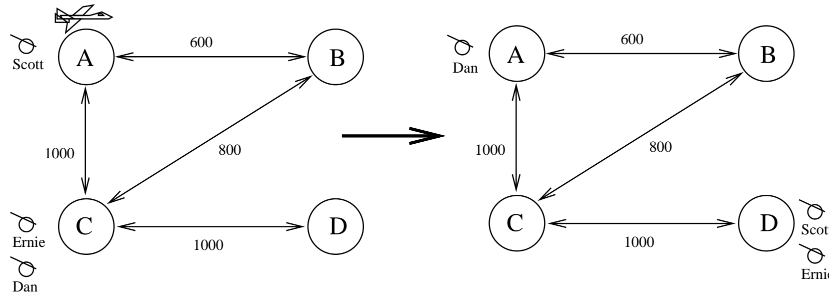


Figure 11.1: An Instance to the *Zeno-Travel* Domain: Start State (left) and Goal State (right).

## 11.3 Terminology

Our running example is the following instance of a simple PDDL 2.1 problem in *Zeno-Travel* and illustrated in Figure 11.1. The initial configuration is drawn to the left of the figure and the goal configuration to its right. Some global and local numerical variable assignment are not shown.

In Figures 11.2 and 11.3 we provide the according textual domain and problem specifications[3]. The instance asks for a temporal plan to fly passengers (`dan`, `scott`, and `ernie`) located somewhere on a small map (including the cities `city-a`, `city-b`, `city-c`, and `city-d`) with an aircraft (`plane`) to their respective target destinations. Boarding and debarking takes a constant amount of time. The plane has a determined capacity of fuel. Fuel and time are consumed according to the distances between the cities and with respect to two different travel speeds. Since fuel can be restored by refueling the aircraft, the total amount of fuel is also maintained as a numerical quantity.

### 11.3.1 Sets and Indices

Table 11.1 displays the basic terminology for sets used in this paper. As most currently successful planning system, MIPS grounds parameterized information present in the domain description.

For all sets we infer a suitable array embedding, indicated by a mapping $\phi$ from this set to a finite domain and vice versa. This embedding is important to deal with unique identifiers of entities instead of their textual or internal representation. The arrays containing the corresponding information can then be accessed in constant time. Almost all planners that perform grounding prior to the search address instantiations by identifiers.

For sets that occur in the domain or problem specification without any parameterization like $\mathcal{CONST}, \mathcal{PRED}, \mathcal{FUNC}, \mathcal{ACT}, \mathcal{TYPE}, \mathcal{ACT}$, and $\mathcal{OBJ}$, the index $\phi$ refers to the position of occurrence. Let $k(p)$, $k(f)$, and $k(a)$ denote the arity (the number of parameters) of predicate $p \in \mathcal{PRED}$, function $f \in \mathcal{FUNC}$, and $a \in \mathcal{ACT}$, respectively. The index for an instantiated predicate $(p\ o_1 \ldots o_{k(p)}) \in \mathcal{IPRED}$ is computed as

[3] `[...]` denotes that source fragments were omitted for the sake of brevity. In the given example these are the action definitions for debarking a passenger and flying an airplane..

```
(define (domain zeno-travel)
(:requirements :durative-actions :typing :fluents)
(:types aircraft person city)
(:predicates (at ?x - (either person aircraft) ?c - city)
             (in ?p - person ?a - aircraft))
(:functions (fuel ?a - aircraft) (distance ?c1 - city ?c2 - city)
            (slow-speed ?a - aircraft) (fast-speed ?a - aircraft)
            (slow-burn ?a - aircraft)  (fast-burn ?a - aircraft)
            (capacity ?a - aircraft)   (refuel-rate ?a - aircraft)
            (total-fuel-used) (boarding-time) (debarking-time))
(:durative-action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :duration (= ?duration boarding-time)
 :condition (and (at start (at ?p ?c))
                 (over all (at ?a ?c)))
 :effect (and (at start (not (at ?p ?c)))
             (at end (in ?p ?a))))
[...]
(:durative-action zoom
 :parameters (?a - aircraft ?c1 ?c2 - city)
 :duration (= ?duration (/ (distance ?c1 ?c2) (fast-speed ?a)))
 :condition (and (at start (at ?a ?c1))
                 (at start (>= (fuel ?a) (* (distance ?c1 ?c2) (fast-burn ?a)))))
 :effect (and (at start (not (at ?a ?c1)))
             (at end (at ?a ?c2))
             (at end (increase total-fuel-used
                        (* (distance ?c1 ?c2) (fast-burn ?a))))
             (at end (decrease (fuel ?a)
                        (* (distance ?c1 ?c2) (fast-burn ?a))))))
(:durative-action refuel
 :parameters (?a - aircraft ?c - city)
 :duration (= ?duration (/ (- (capacity ?a) (fuel ?a)) (refuel-rate ?a)))
 :condition (and (at start (< (fuel ?a) (capacity ?a)))
                 (over all (at ?a ?c)))
 :effect (at end (assign (fuel ?a) (capacity ?a))))
)
```

Figure 11.2: Zeno-Travel Domain Description in PDDL2.1.

$$\phi((p\, o_1 \dots o_{k(p)})) = \psi(p) + \sum_{i=1}^{k(p)} \phi(o_i) |\mathcal{OBJ}|^{i-1},$$

where $\psi(p) = \sum_{i=1}^{\phi(p)-1} |\mathcal{OBJ}|^{k(p_i)}$ is the offset of predicate $p \in \mathcal{PRED}$ and $|\mathcal{OBJ}|$ is the cardinality of set $\mathcal{OBJ}$. Taking $|\mathcal{OBJ}|$ as the radix is a rather coarse value for all parameter instantiations; one could refine the index by using parameter type information.

Indices for instantiated functions $(f\ o_1 \dots o_{k(f)}) \in \mathcal{IFUNC}$ are determined analogously. Instantiated actions $a \in \mathcal{IACT}$ with parameters $p_1, \dots, p_{k(a)}$ are consequently addressed by the following index

$$\phi((a\, p_1 \dots p_{k(a)})) = \psi(a) + \sum_{i=1}^{k(a)} \phi(p_i) |\mathcal{OBJ}|^{i-1}.$$

After static analysis has established a superset of all occurring fluents $\mathcal{F}$, operators $\mathcal{O}$ and variables $\mathcal{V}$, in MIPS the index range is reduced to a minimum, thereby refining $\phi$ to $\phi'$. In the following we keep $\phi$ as a descriptor, and assume that $\phi(p) \in \{1, \dots, |\mathcal{P}|\}$, $\phi(f) \in \{1, \dots, |\mathcal{V}|\}$, and $\phi(a) \in \{1, \dots, |\mathcal{O}|\}$.

```
(define (problem zeno-travel-1)
    (:domain zeno-travel)
    (:objects plane - aircraft
              ernie scott dan - person
              city-a city-b city-c city-d - city)
    (:init (= total-fuel-used 0) (= debarking-time 20) (= boarding-time 30)
           (= (distance city-a city-b) 600)  (= (distance city-b city-a) 600)
           (= (distance city-b city-c) 800)  (= (distance city-c city-b) 800)
           (= (distance city-a city-c) 1000) (= (distance city-c city-a) 1000)
           (= (distance city-c city-d) 1000) (= (distance city-d city-c) 1000)
           (= (fast-speed plane) (/ 600 60)) (= (slow-speed plane) (/ 400 60))
           (= (fuel plane) 750)              (= (capacity plane) 750)
           (= (fast-burn plane) (/ 1 2))     (= (slow-burn plane) (/ 1 3))
           (= (refuel-rate plane) (/ 750 60))
           (at plane city-a) (at scott city-a) (at dan city-c) (at ernie city-c))
    (:goal (and (at dan city-a) (at ernie city-d) (at scott city-d)))
    (:metric minimize total-time)
)
```

Figure 11.3: Zeno-Travel Problem Instance.

| Set | Descriptor | Example(s) |
|---|---|---|
| $\mathcal{OBJ}$ | objects | `dan`, `city-a`, `plane`, ... |
| $\mathcal{TYPE}$ | object types | `aircraft`, `person`, ... |
| $\mathcal{PRED}$ | predicates | `(at ?a ?c)`, `(in ?p ?a)`, ... |
| $\mathcal{FUNC}$ | numerical functions | `(fuel ?a)`, `(total-time)`, ... |
| $\mathcal{ACT}$ | parameterized actions | `(board ?a ?p)`, `(refuel ?a)`, ... |
| $\mathcal{IACT}$ | instantiated actions | `(board plane scott)`, ... |
| $\mathcal{O} \subseteq \mathcal{IACT}$ | fluent operators | `(board plane scott)`, ... |
| $\mathcal{IPRED}$ | instantiated predicates | `(at plane city-b)`, ... |
| $\mathcal{F} \subseteq \mathcal{IPRED}$ | fluents | `(at plane city-b)`, ... |
| $\mathcal{IFUNC}$ | instantiated funtions | `(distance city-a city-b)`, ... |
| $\mathcal{V} \subseteq \mathcal{IFUNC}$ | variables | `(fuel plane)`, `(total-time)`, ... |

Table 11.1: Basic Set Definitions.

In the following we first give the formal description of a grounded planning problem and then turn to the static analyzer that infers the according and supplementary information.

### 11.3.2 Grounded Planning Problem Instances

As many other planners MIPS refers to grounded planning problem representations.

**Definition 10** *(Grounded Planning Instance) A* grounded planning instance *is a quadruple $\mathcal{P} = \langle \mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{G} \rangle$, where $\mathcal{S}$ is the set of planning states, $\mathcal{I} \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states. In mixed propositional and numerical planning problem the state space $\mathcal{S}$ is given by*

$$\mathcal{S} \subseteq 2^{\mathcal{F}} \times I\!R^{|\mathcal{V}|},$$

*where $2^{\mathcal{F}}$ is the power set of $\mathcal{F}$. Therefore, a state $S \in \mathcal{S}$ is a pair $(S_p, S_n)$ with propositional part $S_p \in 2^{\mathcal{F}}$ and numerical part $S_n \in I\!R^{|\mathcal{V}|}$.*

For the sake of brevity, we assume the operators to be in *normal form*, by means that propositional parts (preconditions and effects) satisfy standard STRIPS notation [126] and numerical parts are given in form of arithmetic trees $t$ taken from the set of all trees $T$ with arithmetic operations in the nodes and numerical variables and evaluated constants in the leaves. With *LeafVariables*$(t)$, $t \in T$, we denote the set of all leaf variables in the tree $t$. However, there is no fundamental difference to more general preconditions and effects representations. The current implementation in MIPS takes a generic precondition tree, thereby including comparison symbols, logical operators (in the nodes) and arithmetic subtrees.

**Definition 11** *(Syntax of Grounded Planning Operator) An operator $o \in \mathcal{O}$ in normal form $o = (\alpha, \beta, \gamma, \delta)$ has propositional preconditions $\alpha \subseteq \mathcal{F}$, propositional effects $\beta = (\beta_a, \beta_d) \subseteq \mathcal{F}^2$, numerical preconditions $\gamma$, and numerical effects $\delta$. A* numerical precondition *$c \in \gamma$ is a triple $c = (h_c, \otimes, t_c)$, where $h_c \in \mathcal{V}$, $\otimes \in \{\leq, <, =, >, \geq\}$, and $t_c \in T$. A* numerical effect *$m \in \delta$ is a triple $m = (h_m, \oplus, t_m)$, where $h_m \in \mathcal{V}$, $\oplus \in \{\leftarrow, \uparrow, \downarrow\}$ and $t_m \in T$.*

Obviously, $\otimes \in \{\leq, <, =, >, \geq\}$ represents the associated comparison relation, while $\leftarrow$ denotes an assigment to a variable, while $\uparrow$ and $\downarrow$ indicate a respective increase or decrease operation to it. This allows to formalize the application of planning operators to a given state.

**Definition 12** *(Semantics of Grounded Planning Operator Application) An operator $o = (\alpha, \beta, \gamma, \delta) \in \mathcal{O}$ applied to a state $S = (S_p, S_n)$, $S_p \in 2^{\mathcal{F}}$ and $S_n \in I\!R^{|\mathcal{V}|}$, yields a successor state $S' = (S'_p, S'_n) \in 2^{\mathcal{F}} \times I\!R^{|\mathcal{V}|}$ as follows.*

*We say that a vector $S_n = (S_1, \ldots, S_{|\mathcal{V}|})$ of numerical variables* satisfies *a numerical constraint $c = (h_c, \otimes, t_c) \in \gamma$ if $s_{\phi(h_c)} \otimes eval(S_n, t_c)$ is true, where $eval(S_n, t_c) \in I\!R$ is obtained by substituting all $v \in \mathcal{V}$ in $t_c$ by $S_{\phi(h_c)}$ followed by a simplification of $t_c$.*

*If $\alpha \subseteq S_p$ and $S_n$* satisfies *all $c \in \gamma$ then $S'_p = S_p \cup \beta_a \setminus \beta_d$ and the vector $S_n$ is* updated *for all $m \in \delta$ . We say that the vector $S_n = (S_1, \ldots, S_{|\mathcal{V}|})$ is* updated *to vector $S'_n = (S'_1, \ldots, S'_{|\mathcal{V}|})$ by modifier $m = (h_m, \oplus, t_m) \in \delta$, if*

- *$S'_{\phi(h_m)} = eval(S_n, t_m)$ for $\oplus = \leftarrow$,*
- *$S'_{\phi(h_m)} = S_{\phi(h_m)} + eval(S_n, t_m)$ for $\oplus = \uparrow$, and*
- *$S'_{\phi(h_m)} = S_{\phi(h_m)} - eval(S_n, t_m)$ for $\oplus = \downarrow$.*

The propositional update $S'_p = S_p \cup \beta_a \setminus \beta_d$ is defined as in standard STRIPS. The set of goal states $\mathcal{G}$ is often given as $\mathcal{G} = (\mathcal{G}_p, \mathcal{G}_n)$ with a partial propositional state descrition $\mathcal{G}_p \subset \mathcal{F}$, and $G_n$ as a set of numererical preconditions $c = (h_c, \otimes, t_c)$. Moreover, the arithmetic trees $t_c$ usually collaps to simple leaves labeled with numerical constants. Hence, we might assume that $|\mathcal{G}_n| \leq |\mathcal{V}|$.

### 11.3.3 Static Analysis

The static analyzer takes the domain and problem instance as an input, grounds its propositional state information and infers different forms of planner independent static information.

- Parsing: Our simple Lisp parser generates a tree of Lisp entities. It reads the input files and recognizes the domain and problem name. To cope with typing we temporarily assert constant typed predicates to be removed together with other constant predicates in a further pre-compiling step. Thereby, we infer a type hierarchy and an associated mapping of objects to types.

- Indexing: Based on the number of counted objects, first indices for the grounded predicates, functions and actions are devised. Since in our example problem we have eight objects and the predicates `at` and `in` have two parameters, we reserve $2 \cdot 8 \cdot 8 = 128$ index positions. Similarly, the function `distance` consumes 64 indices, while `fuel`, `slow-speed`, `fast-speed`, `slow-burn`, `fast-burn`, `capacity`, and `refuel-rate` each reserve eight index positions. For the quantities `total-fuel-used`, `boarding-time`, `debarking-time` only a single fact identifier is needed. Last but not least we interpret duration as an additional quantity `total-time`.

- Flattening Temporal Identifiers: According to our assumption of finite branching in this phase we interpret each action as in integral entity, so that all timed propositional and numerical preconditions can be merged. Similarly, all effects are merged, independent of their happening. Invariance conditions like `(over all (at ?a ?c))` in the action `board` are included into the precondition set. We will discuss the rationale of this step in Section 11.5.

- Grounding Propositions: Fact-space exploration is a relaxed enumeration of the planning problem to determine a superset of all reachable facts. Algorithmically, a FIFO fact queue is comprised. Successively extracted facts at the front of the queue are matched to the operators. Each time all preconditions of an operator are fulfilled, the resulting atoms according to the positive effect (add) list are determined and enqueued. This allows to distinguish constant from fluent facts, since only the latter are reached by exploration.

- Grouping Atoms: For a concise encoding of the propositional part we group fluent facts in sets of mutually exclusive groups, so that each state in the planning space can be expressed as a conjunct of (possibly trivial) facts drawn from each fact group [101]. More formally, let $\#p_i(o_1, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n)$ be the number of objects $o_i$ for which the fact $(p\ o_1\ \ldots\ o_n)$ is true. We establish a single-valued invariance at $i$ if $\#p_i(o_1, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n) = 1$. All fix object $o_j$, $j \neq i$, are representative of the invariance and label the group. To allow for a better encoding, some predicates like `at` and `in` are merged. In the example three groups determine the unique position of the persons (one of five) and one group determines the position of the plane (one of four). Therefore, $3 \cdot \lceil \log 5 \rceil + 1 \cdot \lceil \log 4 \rceil = 11$ bits suffice to encode the encountered 19 fluent facts.

- Grounding Actions: Fact-space exploration also determines all grounded operators. Once all preconditions are met and grounded, the symbolic effect lists are instantiated. In our case we determine 98 instantiated operators, which, by some further simplifications that eliminate duplicates and void operators, are reduced to 43.

- Grounding Functions: Synchronous to fact space exploration of the propositional part of the problem all heads of the numerical formulae in the effect lists are grounded. In the example case only three instantiated formulae are fluent: `fuel plane` with initial value 750 as well as `total-fuel-used` and `total-time` both initialized with zero. All other numerical predicates are in fact constants that can be substituted in the formula-bodies. For example, the numerical effect in `board dan city-a` reduces to `(increase (total-time) 30)`, while `zoom plane city-a city-b` has the following numerical effects: `(increase (total-time) 150)`, `(increase (total-fuel-used) 300))`, and `(decrease (fuel plane) 300)`. Refueling, however, does not reduce to a single rational number, e.g. the effects in `refuel plane city-a` only simplify to `(increase (total-time) (/ (- (750 (fuel plane)) / 12.5)))` and `(assign (fuel plane) 750)`. To evaluate the former assignment variable `total-time` has to be instantiated *on-the-fly*. This is due to the fact that the value of the quantity `fuel plane` is not constant and itself changes over time.

## 11.4 Architecture of MIPS

Figure 11.4 depicts the main components of MIPS and the data flow from the input definition of the domain and the problem instance to the resulting temporal plan in the output.

The planning process can be coarsely grouped into two stages, static analysis and (heuristic search) planning.

The intermediate textual format of the static analyzer in annotated grounded PDDL-like representation serves as an interface e.g. for other planners or model checkers and as an additional resource for plan visualization. Figures 11.5 and 11.6 depict an example output for the intermediate representation in the *Zeno-Travel* example.

The object-oriented framework design of MIPS allows different heuristic estimates to be combined with different search strategies, access data structures, and scheduling options.

### 11.4.1 Heuristics

MIPS incorporates more than six different estimates.

- Relaxed planning heuristic (RPH): Approximation of the number of planning steps needed to solve the propositional planning problem with all delete effects removed [181]. The heuristic is constructive, i.e. it returns the set of operators that appear in the relaxed plan.

- Numerical relaxed planning heuristic (numerical RPH): Our numerical extension to RPH is a combined propositional and numerical forward and backward approximation scheme, also allowing for multiple operator application. Our version for integrating numbers into the relaxed planning heuristic is sound, but not as general as Hoffmann's contribution [179]: it restricts to variable-to-constant comparisons, and lacks the simplification of linear constraints.

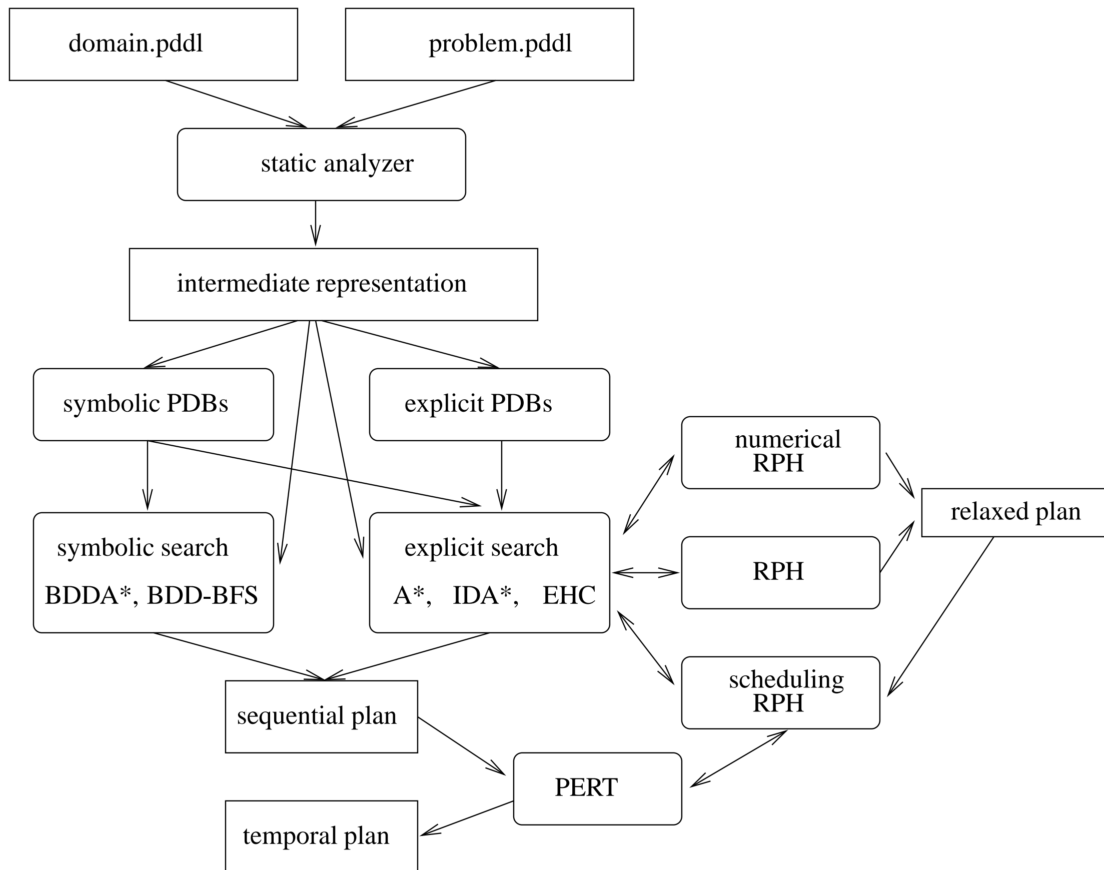


Figure 11.4: Architecture of MIPS

- Pattern databases heuristic (explicit PDB heuristic): Explicit PDBs were already mentioned in the historical overview of MIPS. The different abstractions are found in a greedy best-fit bin-packing manner, yielding a selection of large PDBs in form of perfect hash tables that fit into main memory. If necessary, PDBs can be designed to be disjoint yielding an admissible estimate [95].

- Symbolic pattern database heuristic (symbolic PDB heuristic): Symbolic PDBs apply to both explicit and symbolic heuristic search engines. Due to the succinct BDD-representation of sets of states the averaged heuristic estimate can be increased while decreasing the number of nodes to be explored in the overall search. Symbolic PDBs are often orders of magnitudes larger than explicit ones. Due to state conversion into a Boolean representation the retrieval of a heuristic estimate is slower than hashing, but still linear in the state description length [99],

- Scheduling relaxed plan heuristic (scheduling RPH, SRPH): Critical-path analysis by PERT scheduling may also guide the plan finding phase. Different to the RPH heuristics, which computes the length of the greedily extracted plan, SRPH also takes the sequence of operators into account and searches for a good parallel arrangement. Adding PERT-schedules for the path to a state and for the sequence of actions in the relaxed plan is not as accurate as the PERT-schedule of the combined paths. Therefore, the classical merit function of A*-like search engines $f = g + h$

```
(define (grounded zeno-travel-zeno-travel-1)
  (:fluents
     (at dan city-a)   (at dan city-b)   (at dan city-c)   (at dan city-d)
     (at ernie city-a) (at ernie city-b) (at ernie city-c) (at ernie city-d)
     (at plane city-a) (at plane city-b) (at plane city-c) (at plane city-d)
     (at scott city-a) (at scott city-b) (at scott city-c) (at scott city-d)
     (in dan plane)    (in ernie plane)  (in scott plane))
  (:variables (fuel plane) (total-fuel-used) (total-time))
  (:init
     (at dan city-c)   (at ernie city-c) (at plane city-a) (at scott city-a)
     (= (fuel plane) 750) (= (total-fuel-used) 0) (= (total-time) 0))
  (:goal  (at dan city-a)   (at ernie city-d) (at scott city-d))
  (:metric minimize (total-time) )
  (:group dan
    (at dan city-a)    (at dan city-b)   (at dan city-c)    (at dan city-d)
    (in dan plane))
  (:group ernie
    (at ernie city-a)  (at ernie city-b) (at ernie city-c)  (at ernie city-d)
    (in ernie plane))
  (:group plane
    (at plane city-a)  (at plane city-b) (at plane city-c)  (at plane city-d))
  (:group scott
    (at scott city-a)  (at scott city-b) (at scott city-c)  (at scott city-d)
    (in scott plane))
```

Figure 11.5: Grounded Representation of *Zeno-Travel* Domain.

of generating path length $g$ and heuristic estimate $h$ is not immediate. We define the heuristic value of SRPH as the parallel plan length of the combined path minus the parallel plan length of the generating path.

- One suitable combination of the PDB heuristic and RPH heuristics that is also implemented in MIPS, compares the retrieved result of the PDBs with the set of operators in the plan graph that respect the abstraction. The intuition is to slice the relaxed plan graph. If in the backward exploration an add-effect is selected the match will be assigned to its fact group. If the number of matches in an abstraction is smaller than the retrieved PDB value it will be increased by the lacking amount.

In the competition, except for numerical domains we chose pure RPH for sequential plan generation and scheduling PRH for temporal domains. Only in pure numerical problems we used numerical RPH. We have experimented with (symbolic) PDBs with mixed results. Since in our implementation PDBs are purely propositional and do not allow the retrieval of the corresponding operator sets of the optimal abstract plan, we have not included PDB search in the competition version of MIPS.

### 11.4.2 Exploration Algorithms

The algorithm portfolio includes:

- Weighted A* (weighted A*/A*): The A* algorithm [161] can be casted as a derivate of Dijkstra's SSSP exploration on a re-weighted graph. For lower bound heuristics, original A* can be shown to generate optimal plans [287]. Weightening the influence of the heuristic estimate may accelerate solution finding, but also affects optimality [295]. The set of horizon nodes are maintained in a priority queue *Open*, while the settled nodes are kept in *Closed*.

```
  (:action board dan plane city-a
   :condition
     (and (at dan city-a) (at plane city-a))
   :effect
     (and (in dan plane)  (not (at dan city-a))
        (increase (total-time) (30.000000))))
  [...]
  (:action zoom plane city-a city-b
   :condition
     (and
        (at plane city-a)
        (>= (fuel plane) (300.000000)))
   :effect
     (and (at plane city-b)  (not (at plane city-a))
        (increase (total-time) (60.000000))
        (increase (total-fuel-used) (300.000000))
        (decrease (fuel plane) (300.000000))))
  [...]
  (:action refuel plane city-a
   :condition
     (and
        (at plane city-a)
        (< (fuel plane) (750.000000)))
   :effect
     (and
        (increase (total-time) (/ (- (750.000000) (fuel plane)) (12.500000)))
        (assign (fuel plane) (750.000000))))
 [...]
)
```

Figure 11.6: Grounded Representation of *Zeno-Travel* Domain (cont.).

In MIPS, Weighted A* is implemented with a Dial or a Weak-Heap priority queue data structure [79, 116]. The former is used for propositional planning only, while the latter applies to general planning with scheduling estimates. Arrays have been implemented as a dynamic table that double their sizes if they become filled. MIPS stores all generated and expanded states in a hash table. An alternative, yet not implemented, but more flexible storage structure is collection of persistent trees as in the TL planning system [17], one for each predicate. In the best case queries and update times to the structure are logarithmic in the number of represented atoms.

- Weighted Iterative-Deepening A* ((W)IDA*): The memory-limited variant of (Weighted) A* is well-suited to large exploration problems with efficient evaluation functions of small integer range [213]. In MIPS, IDA* is extended with bit-state hashing [111] to improve duplicate detection with respect to ordinary transposition tables [305]. This form of partial search effectively trades state-space coverage for completeness. For a further compression of the planning state space, all variables that appear in the objective function are neglected from hash address calculations and state comparisons.

- Enforced Hill Climbing (EHC): The approach is another compromise between exploration and exploitation. EHC searches with an improved evaluation in breadth-first manner and commits established decisions as final [177]. EHC is complete in undirected problem graphs and seems to have a slight advantage to Weighted A* when combined with RPH and other pruning cuts. On the other hand, it can be misguided in unstructured planning domains and is likely to get lost in problem graphs with dead-ends.

- Bidirectional Symbolic Breadth-First-Search (BDD-BFS): The implementation performs bidirectional blind symbolic search, choosing the next search direction in favor to the faster executions of the previous iterations [101].

- Weighted Symbolic A* (BDDA*): The algorithm performs guided symbolic search and takes a (possibly partitioned) symbolic representation of the heuristic as an additional input. Given a consistent estimate for a uniformly weighted graph, BDDA* performs at most $\mathcal{O}(f^{*^2})$ iterations, where $f^*$ is the optimal solution length, where consistent estimates keep the accumulated $f$-values on each exploration path monotonical increasing.

- Weak and Strong Planning: These two symbolic exploration algorithms suited to non-deterministic planning have been added to MIPS [63], but due to the lack of an agreed standard for a domain description language, the implementation was only tested on deterministic samples in which the above symbolic algorithms clearly perform better. The encoding scheme directly transfers to the non-deterministic scenario, where plans were stored in a form of state-action tables.

In the competition we applied Weighted A* with weight 2, e.g. the merit for all states $S \in \mathcal{S}$ was fixed as $f(S) = g(S) + 2 \cdot h(S)$, yielding good but not necessarily optimal plans. In temporal domains we introduced an additional parameter $\delta$ to scale the influence between propositional estimates ($f_p(S) = g_p(S) + 2 \cdot h_p(S)$) and scheduled ones ($f_s(S) = g_s(S) + 2 \cdot h_s(S)$). More precisely, we altered the comparison function for the priority queue, so that a comparison of parallel length priorities was invoked if the propositional difference of values was not larger than $\delta \in \mathbb{N}_0$. A higher value of $\delta$ refers to a higher influence of the SRPH, while $\delta = 0$ indicates no scheduling at all. In the competition we produced data with $\delta = 0$ (Pure MIPS), and $\delta = 2$ (optimized MIPS).

# 11.5 Temporal Planning

PDDL 2.1 domain descriptions include temporal modifiers *at start*, *over all*, and *at end*, where label *at start* denotes the preconditions and effects at invokation time of the action, *over all* refers to an invariance condition and *at end* to the finalization conditions and consequences of the action.

## 11.5.1 Temporal Model

In Figure 11.7 we show two different options to flatten this information back to planning with preconditions and effects to derive its semantic.

In the first case (top right), the compound operator is split into three smaller parts, one for action invocation, one for invariance maintenance, and one for action termination. This is the semantic suggested by [131].

As expected there are no effects in the invariance pattern, i.e. $B' = \emptyset$. Moreover, we found that in the benchmarks it is uncommon that new effects in `at-start` are preconditioned for termination control or invariance maintenance, i.e. $A' \cap (B \cup C) = \emptyset$.

Therefore, in MIPS the simpler second operator representation model was chosen (bottom right). The intermediate format of the example problem in Figures 11.5 and 11.6

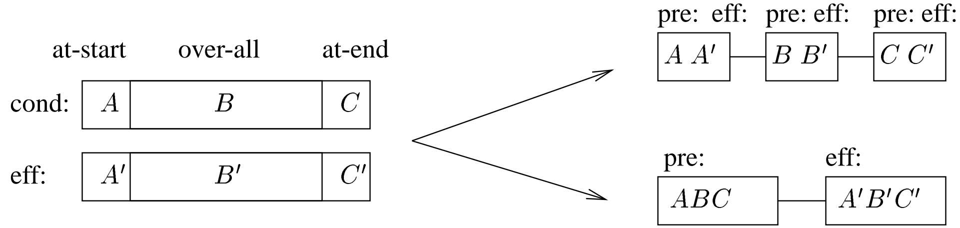


Figure 11.7: Compiling Temporal Modifiers into Operators.

implicitly assumed this simpler temporal model. At least for sequential plan finding we have not observed any deficiencies by assuming this temporal model, in which each action starts immediately after a previous one has terminated.

This simple temporal model motivates the definition of the first plan objective: the sequential plan.

**Definition 13** *(Sequential Plan) A* sequential plan $\pi_s = (O_1, \ldots, O_k)$ *is an ordered sequence of operators* $O_i \in \mathcal{O}$, $i \in \{1, \ldots, k\}$, *that transforms the initial state* $\mathcal{I}$ *into one of the goal states* $G \in \mathcal{G}$, *i.e., there exists a sequence of states* $S_i \in \mathcal{S}$, $i \in \{0, \ldots, k\}$, *with* $S_0 = \mathcal{I}$, $S_k = G$ *and* $S_i$ *is the outcome of applying* $O_i$ *to* $S_{i-1}$, $i \in \{1, \ldots, k\}$.

Minimizing sequential plan length was the only objective in the first and second planning competition. Since *Graphplan*-like planners [39] like IPP [212] and STAN [243] already produced parallel plans, this was a indeed a limiting aspect to evaluate plan quality. The most important reason for this artificial restriction was that total-ordered plans were easier accessible for automated validation, a necessity for evaluating correctness in a competitive scenario.

### 11.5.2 Operator Dependency

The formal definition of operator dependency allows to avoid the transpositioned generation of independent actions and, more importantly, enables optimal schedules of sequential plans with respect to the generated action sequence and its causal structure. If all operators are dependent (or void with respect to the optimizer function), the problem is inherent sequential and no schedule leads to any improvement.

**Definition 14** *(Dependency/Mutex Relation) Two grounded operators* $o = (\alpha, \beta, \gamma, \delta)$ *and* $o' = (\alpha', \beta', \gamma', \delta')$ *in* $\mathcal{O}$ *are* dependent/mutex*, if one of the following three conditions holds:*

1. *The propositional precondition set of one operator has a non-empty intersection with the add or the delete lists of the other one, i.e.,* $\alpha \cap (\beta'_a \cup \beta'_d) \neq \emptyset$ *or* $(\beta_a \cup \beta_d) \cap \alpha' \neq \emptyset$.
2. *The head of a numerical modifier of one operator is contained in some condition of the other one, i.e. there exists a* $c' = (h'_c, \otimes, t'_c) \in \gamma'$ *and a* $m = (h_m, \oplus, t_m) \in \delta$ *with* $h_m \in \text{LeafVariables}(t'_c) \cup \{h'_c\}$ *or there exists a* $c = (h_c, \otimes, t_c) \in \gamma$ *and a* $m' = (h'_m, \oplus, t'_m) \in \delta'$ *with* $h'_m \in \text{LeafVariables}(t_c) \cup \{h_c\}$. *Intuitively, an operator modifies variables that appear in the condition of the other. This may be referred to as a* direct conflict.

3. *The head of the numerical modifier of one operator is contained in the formula body of the modifier of the other one, i.e., there exists a* $m = (h_m, \oplus, t_m) \in \delta$ *and* $m' = (h'_m, \oplus, t'_m) \in \delta'$ *with* $h_m \in \text{LeafVariables}(t'_m)$ *or* $h'_m \in \text{LeafVariables}(t_m)$. *This may be referred to as an* indirect conflict.

The dependence relation may be refined according to the PDDL 2.1 guidelines for mutual exclusion [131], but for our purposes for improving sequential plans this approach is sufficient. In our implementation (at least for temporal and numerical planning) the dependence relation is computed beforehand and tabulated for constant time access. To improve the efficiency of pre-computation, the set of leaf variables is maintained in an array, once the grounded operator is constructed.

To detect domains for which any parallelization leads to no improvement, a planning domain is said to be *inherently sequential* if all operators in any sequential plan are dependent or instantaneous (i.e. with zero duration). The static analyzer checks this by testing each operator pair. While some benchmark domains like *Desert-Rats* and *Jugs-and-Water* are inherently sequential, others like *Zeno-Travel* and *Taxi* are not.

Operator independence also indicates transpositions of two operators $o_1$ and $o_2$ to safely prune exploration in sequential plan generation.

**Definition 15** *(Parallel Plan) A* parallel plan $\pi_c = ((O_1, t_1), \ldots, (O_k, t_k))$ *is a* schedule *of operators* $O_i \in \mathcal{O}$, $i \in \{1, \ldots, k\}$, *that transforms the initial state* $\mathcal{I}$ *into one of the goal states* $G \in \mathcal{G}$, *where* $O_i$ *is executed at time* $t_i$.

[18] clearly distincts partial ordered plans $(O_1, \ldots, O_k, \preceq)$, with the relation $\preceq \subseteq \{O_1, \ldots, O_k\}^2$ being a partial order (reflexive, transitive, and antisymmetric), from parallel plans $(O_1, \ldots, O_k, \preceq, \#)$, with $\# \subseteq (\preceq \cup \preceq^{-1})$ (irreflexive, symmetric) expressing, which actions must not be executed in parallel.

**Definition 16** *(Precedence Ordering) A ordering* $\preceq_d$ *induced by the set of operators* $\{O_1, \ldots, O_k\}$ *and a dependency relation is given by* $O_i \preceq_d O_j$, *if* $O_i$ *and* $O_j$ *are dependent and* $1 \leq i < j \leq k$.

Precedence is not a partial ordering, since it is neither reflexive nor transitive. By computing the transitive closure of the relation, however, precedence could be extended to a partial ordering. A sequential plan $O_1, \ldots, O_k$ produces an acyclic set of precedence constraints $O_i \preceq_d O_j$, $1 \leq i < j \leq k$, on the set of operators. It is also important to observe, that the constraints are already topologically sorted according to $\preceq_d$ by taking the node ordering $\{1, \ldots, k\}$.

**Definition 17** *(Respecting Precedence Ordering in Parallel Plan) Let* $d(O)$ *for* $O \in \mathcal{O}$ *be the* duration *of operator* $O$ *in a sequential plan. For a parallel plan* $\pi_c = ((O_1, t_1), \ldots, (O_k, t_k))$ *that respect* $\preceq_d$, *we have* $t_i + d(O_i) \leq t_j$ *for* $O_i \preceq_d O_j$, $1 \leq i < j \leq k$.

For optimizing plans [18] defines *parallel execution time* as $\max\{t_i + d(O_i) \mid O_i \in \{O_1, \ldots, O_k\}\}$, so that if $O_i \preceq O_j$, then $t_i + d(O_i) \leq t_j$, and if $O_i \# O_j$, then either $t_i + d(O_i) \leq t_j$ or $t_j + d(O_j) \leq t_i$. These two possible choices in $\#$ are actually not apparent in practice, since we already have a precedence relation at hand and just seek

**Procedure** *Critical-Path*
**Input:** *Sequence of operators* $O_1, \ldots, O_k$, precedence ordering $\preceq_d$
**Output:** Optimal parallel plan length $\max\{t_i + d(O_i) \mid O_i \in \{O_1, \ldots, O_k\}\}$

```
for all i ∈ {1,...,k}
    e(O_i) = d(O_i)
    for all j ∈ {1,...,i-1}
        if (O_j ⪯_d O_i)
            if e(O_i) < e(O_j) + d(O_i)
                e(O_i) ← e(O_j) + d(O_i)
return max_{1≤i≤k} e(O_i)
```

Table 11.2: Algorithm to Compute Critical Path Length.

the optimal arrangement of operators. In contrast we assert that only one option, namely $t_i + d(O_i) \leq t_j$ can be true, reducing $\#$ to $\preceq_d$. More importantly, [18]'s work introduces unnecessary time complexity, since optimized scheduling a set of fixed-timed operators is already an NP-complete problem.

**Definition 18** *(Optimal Parallel Plan) An optimal parallel plan with respect to a sequence of operators* $O_1, \ldots, O_k$ *and precedence ordering* $\preceq_d$ *is a parallel plan* $\pi^* = ((O_1, t_1), \ldots, (O_k, t_k))$ *with minimal parallel execution time* $OPT = \max\{t_i + d(O_i) \mid O_i \in \{O_1, \ldots, O_k\}\}$ *among all parallel plans* $\pi_c = ((O_1, t'_1), \ldots, (O_k, t'_k))$ *that respect* $\preceq_d$.

Many algorithms have been suggested to convert sequential plans into partial ordered ones [288, 304, 344]. Most of them interpret a total ordered plan as a maximal constrained partial ordering $\preceq = \{(O_1, O_j) \mid 1 \leq i < j \leq k\}$ and search for least constraint plans. However, the problem of minimum constraint "deordering" has also been proven to be NP-hard, except if the so-called validity check is polynomial [18], where deordering maintains validity of the plan by lessening its constraintness, i.e. $\preceq' \subseteq \preceq$ for a new ordering $\preceq'$.

Since we have an explicit model of dependency and time, optimal parallel plans will not change the ordering relation $\preceq_d$ at all.

### 11.5.3 Critical Path Analysis

The *Project Evaluation and Review Technique* (PERT) is a critical path analysis algorithm usually applied to project management problems. The critical path is established, when the total time for activities on this path is greater than any other path of operators. A delay in any tasks on the critical path leads to a delay in the project. The heart of PERT is a network of tasks needed to complete a project, showing the order in which the tasks need to be completed and their dependencies between them. As shown in Table 11.2, PERT scheduling reduces to a derivate of Dijkstra's single shortest path algorithm within acyclic graphs [70].

In the algorithm, $e(O_i)$ is the tentative earliest end time of operator $O_i$, $i \in \{1, \ldots, k\}$, while the earliest starting times $t_i$ for all operators in the optimal plan are given by $t_i = e(O_i) - d(O_i)$.

**Theorem 4** *(PERT Scheduling) Given a sequence of operators $O_1, \ldots, O_k$ and a precedence ordering $\preceq_d$ an optimal parallel plan $\pi^* = ((O_1, t_1), \ldots, (O_k, t_k))$ can be computed in optimal time $\mathcal{O}(k + | \preceq_d |)$.*

**Proof:** The proof is done by induction on $i \in \{1, \ldots, k\}$. The induction hypothesis is that after iteration $i$ the value $e(O_i)$ is correct, e.g. $e(O_i)$ is the earliest end time of operator $O_i$. This is clearly true for $i = 1$, since $e(O_1) = d(O_1)$. We now assume that the hypothesis is true $1 \leq j < i$ and look at iteration $i$. There are two choices. Either there is a $j \in \{1, \ldots, i-1\}$ with $(O_j \preceq_d O_i)$. For this case after the inner loop is completed, $e(O_i)$ is set to $\min\{e(O_j) + d(O_j) \mid O_j \preceq_d O_i, j \in \{1, \ldots, i-1\}\}$. On the other hand, $e(O_i)$ is optimal, since $O_i$ cannot start earlier than $\min\{e(O_j) \mid O_j \preceq_d O_i, j \in \{1, \ldots, i-1\}\}$, since all values $e(O_j)$ are already the smallest possible by induction hypothesis. If there is no $j \in \{1, \ldots, i-1\}$ with $(O_j \preceq_d O_i)$, then $e(O_i) = d(O_i)$ as in the base case. Therefore, at the end $\max_{1 \leq i \leq k} e(O_i)$ is the optimal parallel path length.

The time and space complexities of the algorithm *Critical-Path* are clearly in $\mathcal{O}(k^2)$, where $k$ is the length of the sequential plan. Using an adjacency list representation these efforts can be reduced to time and space proportional to the number of vertices and edges in the dependence graph, which are of size $\mathcal{O}(k + | \preceq_d |)$. The bound is optimal, since the input consists of $\Theta(k)$ operators and $\Theta(| \preceq_d |)$ dependencies among them. ∎

### 11.5.4 Graphplan Distances

In this section we restrict the planning model to *valid STRIPS plans* as in the original article of *Graphplan* [39], where the execution cost of each operator is 1 and the semantics of a parallel plan are as follows.

For each time step $i$, $i \in \{1, \ldots, l\}$, a state $S_i \in \mathcal{S}$ is generated by applying all operators with time stamp $i - 1$ to $S_{i-1}$, where $S_0 = \mathcal{I}$. An optimal parallel plan is a parallel plan of minimal length $l$. The name *dependency* is borrowed from the notion of partial order reduction in explicit-state model checking [65], where two operators $O_1$ and $O_2$ are *independent* if for each state $S \in \mathcal{S}$ the following two properties hold:

1. *Enabledness* is preserved, i.e. $O_1$ and $O_2$ do not disable each other.

2. $O_1$ and $O_2$ are *commutative*, i.e. executed in any order $O_1$ and $O_2$ lead to the same state.

Two actions *interfere*, if they are dependent. The original *Graphplan* definition is very closed to ours, which fixes interference as $\beta'_d \cap (\beta_a \cup \alpha) \neq \emptyset$ and $(\beta'_a \cup \alpha') \cap \beta_d \neq \emptyset$.

**Lemma 5** *If $\beta_d \subseteq \alpha$ and $\beta'_d \subseteq \alpha'$, operator inference in the* Graphplan *model is implied by the propositional MIPS model of dependence.*

**Proof:** If $\beta_d \subseteq \alpha$ and $\beta'_d \subseteq \alpha'$, for two independent operators $o = (\alpha, \beta)$ and $o' = (\alpha', \beta')$: $\alpha \cap (\beta'_a \cup \beta'_d) = \emptyset$ implies $\beta_d \cap (\beta'_a \cup \beta'_d) = \emptyset$, which in turn yields $\beta_a \cap \beta'_d = \emptyset$. The condition $\beta'_a \cap \beta_d = \emptyset$ can be inferred analogously by exchanging primed and unprimed variables. ∎

**Theorem 5** *Two independent STRIPS operators $o = (\alpha, \beta)$ and $o' = (\alpha', \beta')$ in $\mathcal{O}$ with $\beta_d \subseteq \alpha$ and $\beta'_d \subseteq \alpha'$ are* enabledness preserving *and* commutative*, i.e. for all states in $S \subseteq 2^{|A|}$ we have $o(o'(S)) = o'(o(S))$.*

**Proof:** Since $\beta_d \subseteq \alpha$ and $\beta'_d \subseteq \alpha'$, we have $\beta_a \cap \beta'_d = \emptyset$ and $\beta'_a \cap \beta_d = \emptyset$ by Lemma 5. Let $S'$ be the state $((S \setminus \beta_d) \cup \beta_a)$ and let $S''$ be the state $((S \setminus \beta'_d) \cup \beta'_a)$. Since $(\beta'_a \cup \beta'_b) \cap \alpha = \emptyset$, $o$ is enabled in $S''$, and since $(\beta_a \cup \beta_b) \cap \alpha' = \emptyset$, $o'$ is enabled in $S'$. Moreover,

$$
\begin{aligned}
o(o'(S)) &= (((S \setminus \beta'_d) \cup \beta'_a) \setminus \beta_d) \cup \beta_a \\
&= (((S \setminus \beta'_d) \setminus \beta_d) \cup \beta'_a) \cup \beta_a \\
&= S \setminus (\beta'_d \cup \beta_d) \cup (\beta'_a \cup \beta_a) \\
&= S \setminus (\beta_d \cup \beta'_d) \cup (\beta_a \cup \beta'_a) \\
&= (((S \setminus \beta_d) \setminus \beta'_d) \cup \beta_a) \cup \beta'_a \\
&= (((S \setminus \beta_d) \cup \beta_a) \setminus \beta'_d) \cup \beta'_a = o'(o(S))
\end{aligned}
$$

■

A less restrictive notion of independence, in which several actions may occur at the same time even if one deletes an add-effect of another is provided in [209].

All three models of valid plans are restrictive, since they assume that for each parallel plan there exist at least one corresponding total ordered plan. In general, however, this is not true. Consider the simple STRIPS planning problem domain with $\mathcal{I} = \{B\}$, $\mathcal{G} = \{\{A, C\}\}$, and $\mathcal{O} = \{(\{B\}, \{A\}, \{B\}), (\{B\}, \{C\}, \{B\})\}$. Obviously, both operators are needed for goal achievement, but there is no sequential plan of length 2, since $B$ is deleted in both operators. However, a parallel plan could be executed, since all precondition are fulfilled at the first time step.

### 11.5.5 Full Enumeration Algorithms

Even though full state-space enumaration is far from being practical they provide a basis for heurisitic search engines. In optimal parallel plans, each operator either starts or ends at the start or end time of another operator. Therefore, for a fixed number of operators, we can assume a possibly exponential but finite number of possible parallel plans.

This immediately leads to the following plan enumeration algorithm *ENUM-1*. For all $|\mathcal{O}|^i$ operator sequences of length $i$, $i \in \mathbb{N}$, generate all possible partial orderings, check for each individual schedule if it transforms the initial state into one of the goals, and take the sequence with smallest parallel plan length. Since all parallelizations are computed we have established the following result.

**Theorem 6** *If the number of operators for an optimal parallel plan is bounded, ENUM-1 is complete and computes optimal parallel plans.*

Note that the first $i$ with a matching solution does not necessarily yield an optimal parallel path, since longer operator sequences might rise better parallel solutions. ENUM-1 can also generate non-valid plans in the *Graphplan* model. For a better distinction between the objectives for parallel plans, we keep the notion of validity in this section.

Assuming only valid plans implies that each parallel plan corresponds to at least one (possible many) sequential ones. Viewed from the opposite side, each partial-ordered plan can be established by generating a totally-ordered plan first and then apply a scheduling algorithm to it to find its best partial-order. Therefore, the next two enumeration schemes produce valid plans only.

Enumeration algorithm *ENUM-2* generates all feasible sequential plans of length $i$ with increasing $i \in \mathbb{N}$, and computes their optimal schedule with respect to the number of operators and dependency property. Since optimal parallelization of all valid operator sequences are computed we have established the following theorem.

**Theorem 7** *If the number of operators for an optimal parallel plan is bounded, ENUM-2 is complete and computes a valid optimal parallel plan.*

A complete enumeration scheme of all sequential plans that transform the initial state into one goal state is also still computationally expensive, but ruling out impossible operator applications drastically reduces the vast number of $|\mathcal{O}|^i$ operator sequences of length $i$. Bäckström's result for deriving partial orders has shown, that given the sequence of operators in a sequential plan, to infer an optimized partial order that respects a set of mutexes is NP-hard, so that even for the second phase no polynomial-time algorithm is to be expected. Therefore, at least for STRIPS we have restricted the PSPACE-hard planning task [53] to an NP-hard problem for each generated sequential plan.

When the concept of mutual exclusion is extended to a precedence relation between operators, there exist at least one sequential plan that respects the set of operators and the set of precedence constraints. From the opposite point of view, for each sequential plan there exist at least one parallel plan that respects both the number of operators and the imposed set of precedence constraints.

Algorithm *ENUM-3* is a straight variant of *ENUM-2* that simply applies PERT scheduling for finding the optimal parallel plan, with the main difference that it additionally maintains the causal structure.

We have seen that *ENUM-1* may generate parallel plans that *ENUM-2* cannot produce. Are there also valid plans that *ENUM-2* can produce, but *ENUM-3* cannot? The answer is no. If *ENUM-2* terminates with an optimal schedule, we generate a corresponding sequential plan while preserving the causal structure. Optimal PERT-scheduling of this plan with respect to the set of operators and the imposed precedence relation will yield back the optimal parallel plan. Since all sequential paths are eventually generated, the given partial will also be found by *ENUM-3*. This proves the following result.

**Theorem 8** *If the number of operators for an optimal parallel plan is bounded, ENUM-3 is complete and computes a valid optimal parallel plan.*

In the following, we interpret optimized parallel plans as nodes in a weighted directed graph $G = (V, E, w)$. Edges correspond to possible extensions of the plans with an additional operator, which can be found by a sequentialization of the parallel plan followed by a PERT scheduling operation. The weight function denotes the difference in parallel plan length. Since the set of operators and the precedence set is enlarged, all weights will be greater than or equal to 0. If only a finite number of actions can be executed in parallel, then any infinite path in $G$ has unbounded cost. Therefore, we can traverse $G$ in shortest path ordering using Dijkstra's algorithm to finally yield an optimal parallel plan.

The argument for optimality is that Dijkstra's algorithm is complete, i.e., it cannot exit with failure, since if the horizon list becomes becomes empty there is no solution at all. If the horizon is not empty, there is at least one node on an optimal solution path, which has to be selected before any goal node with larger cost.

**Theorem 9** *If only a finite number of actions can be executed in parallel, Dijkstra's shortest path enumeration is complete and computes a valid optimal parallel plan.*

### 11.5.6 Heuristic Search Enumeration

The enumeration algorithms in the previous section are sound, complete and optimal in theory. On the other hand enumeration schemes do not contradict known undecidability results in numerical planning [170]. If we have no additional information like a bound to the maximal number of actions in a plan or on the number of actions that can be executed in parallel, we cannot say if the enumeration will terminate or not.

The main drawback of the above approaches is that they are seemingly too slow for practical planning. Heuristic search algorithms like A* and IDA* reorder the traversal of states in the planning problem, and an admissible estimate does not affect completeness and optimality. The reason for completeness in finite graphs is that the number of acyclic paths in $G$ is finite and with every node expansion, A* adds new links to its traversal tree. Each newly added link represents a new acyclic path, so that the reservoir of path must eventually be exhausted. The argument is valid for any best-first strategy that prunes cyclic paths, but by their move-committing nature, hill-climbing algorithms are not complete.

[287] has shown that A* is complete even on infinte graphs, demanding that the cost of every infinite path is unbounded. A deeper investigation shows that given an admissible estimate there must always be a node in the current search horizon with optimal priority. Actually to preserve this condition for admissible but not necessarily consistent estimates, already expanded node may have to be reconsidered (re-opening). Hence, A* must also terminate with an optimal solution.

**Theorem 10** *If the cost of every infinite plan is unbounded, A* enumeration with an admissible parallel plan length estimate computes a valid optimal parallel plan.*

Note that the assumption of unbounded sequential plan costs is not true in all benchmark problems, since there may be an infinite sequence of instantaneous events that do not contribute to the plan objective. For example, loading and unloading tanks in *Desert-Rats* does not affect `total-fuel` consumption, which is to be minimized in one benchmark instance.

As a matter of fact, informative admissible parallel plan length estimates are not easy to obtain. This was the reason in MIPS to chose sequential plan generation first, because very effective heuristics are known to generate sequential plans quickly. With the SRPH we choose a parallel plan length approximation, but since it extends PRH, it is known to be not admissible.

```
  0: (zoom plane city-a city-c) [100]          0: (zoom plane city-a city-c) [100]
100: (board dan plane city-c) [30]           100: (board dan plane city-c) [30]
130: (board ernie plane city-c) [30]         100: (board ernie plane city-c) [30]
160: (refuel plane city-c) [40]              100: (refuel plane city-c) [40]
200: (zoom plane city-c city-a) [100]        140: (zoom plane city-c city-a) [100]
300: (debark dan plane city-a) [20]          240: (debark dan plane city-a) [20]
320: (board scott plane city-a) [30]         240: (board scott plane city-a) [30]
350: (refuel plane city-a) [40]              240: (refuel plane city-a) [40]
390: (zoom plane city-a city-c) [100]        280: (zoom plane city-a city-c) [100]
490: (refuel plane city-c) [40]              380: (refuel plane city-c) [40]
530: (zoom plane city-c city-d) [100]        420: (zoom plane city-c city-d) [100]
630: (debark ernie plane city-d) [20]        520: (debark ernie plane city-d) [20]
650: (debark scott plane city-d) [20]        520: (debark scott plane city-d) [20]
```

Figure 11.8: A Sequential Plan for *Zeno-Travel* (left) and its PERT Schedule (right).

### 11.5.7 Pruning Anomalies

Other acceleration techniques like sequential plan hashing, symmetry and transposition cuts have to be chosen carefully to maintain parallel plan length optimality.

Take for example sequential state memorization, i.e. the memorization of states in the sequential plan generation process. This approach does affect parallel optimality, as the following example shows.

Consider the sequences

```
(zoom city-a city-c plane),(board dan plane),(refuel plane),
(zoom city-c city-a plane), (board scott), (debark dan),
(refuel plane),
```

and

```
(board scott),(zoom city-a city-c plane),(board dan plane),
(refuel plane), (zoom city-c city-a plane), (debark dan),
(refuel plane)
```

in the *Zeno-Travel* problem. The set of operators is the same and so is the resulting (sequentially generated) state.

However, the PERT schedule for the first sequence is shorter than the schedule for the second one, since in the previous case the time for boarding `scott` is compensated by the remaining two operators.

For small problems, such anomalies can be avoided by avoided duplicate pruning at all. As an example Figure 11.8 depicts a sequential plan for the example problem instance and its PERT schedule, which turns out to be the overall optimal parallel plan.

In order to generate sequential solutions for large planning problem instances, in the competition version of MIPS we have introduced cuts that affect optimality but reduce the number of expansions significantly.

### 11.5.8 Arbitrary Plan Objectives

In PDDL 2.1 different plan metrics can be devised. In Figure 11.9 we depict two plans found by MIPS when modifying the objective function from minimizing `total-time` to minimize `total-fuel-used`, and to minimize the compound `(+ (* 10 (total-time)) (* 1 (total-fuel-used)))`.

For the first case we computed an optimal value of 1,333.33, while for the second case we established 7,666.67 as the optimized merit. When optimizing time, the ordering of

```
     0: (board scott plane city-a) [30]
    30: (fly plane city-a city-c) [150]
   180: (board ernie plane city-c) [30]
   180: (board dan plane city-c) [30]
   210: (fly plane city-c city-a) [150]
   360: (debark dan plane city-a) [20]
   360: (refuel plane city-a) [53.33]
413.33: (fly plane city-a city-c) [150]
563.33: (fly plane city-c city-d) [150]
713.33: (debark ernie plane city-d) [20]
713.33: (debark scott plane city-d) [20]
```

```
  0: (zoom plane city-a city-c) [100]
100: (board dan plane city-c) [30]
100: (board ernie plane city-c) [30]
100: (refuel plane city-c) [40]
140: (zoom plane city-c city-a) [100]
240: (debark dan plane city-a) [20]
240: (board scott plane city-a) [30]
240: (refuel plane city-a) [40]
280: (fly plane city-a city-c) [150]
430: (fly plane city-c city-d) [150]
580: (debark ernie plane city-d) [20]
580: (debark scott plane city-d) [20]
```

Figure 11.9: Optimized Plans in *Zeno-Travel* according to different Plan Objectives.

board and zoom actions is important. When optimizing *total-fuel* we reduce speed to save fuel consumption to 333.33 per flight but we may board the first passenger immediately. We also save two refuel actions with respect to the first case.

When increasing the importance of time we can trade refueling actions for time, so that both zooming and flight actions are chosen for the complex minimization criterion.

We first thought, that we could simply substitute the plan objective in the PERT scheduling process. However, the results did not match with the ones produced by the validator [245], in which the final time is substituted in the objective function after the plan has been build.

The way we evaluate objective functions that include time is as follows. First we schedule the (relaxed or final) sequential plan. Then we temporarily substitute the `total-time` value in the state with the parallel plan length and evaluate the formula to get the objective function value. To avoid conflicts in subsequent expansions, afterwards we set the value `total-time` back to the optimal one in the sequential plan.

## 11.6 Symmetry

An important feature of parameterized predicates, functions and action descriptions in the domain specification file is that actions are transparent to different bindings of parameters to objects. Disambiguating information is present in the problem instance file.

In case of typed domains, many planners, including MIPS, compile all type information into additional predicates, attach additional preconditions to actions and enrich the initial states by suitable object-to-type atoms.

As a consequence, a symmetry is viewed as a permutation of objects that is present in the current state, in the goal representation, and transparent to the set of operators.

There are $n!$, $n = |\mathcal{OBJ}|$, possible permutations of the set of objects. Taking into account all type information reduces the number of all possible permutation to

$$\binom{n}{t_1, \ldots, t_k} = \frac{n!}{t_1! \cdot \ldots \cdot t_k!}.$$

where $t_i$ is the number of objects with type $i$, $i \in \{1, \ldots, k = |\mathcal{TYPES}|\}$. In a moderate sized logistic domain with 10 cities, 10 trucks, 5 airplanes, and 15 packages, this results in $40!/(10! \cdot 10! \cdot 5! \cdot 15!) \geq 10^{20}$ permutations.

To reduce the number of potential symmetries to a tractable size we restrict symmetries to object transpositions, for which we have at most $n(n-1)/2 \in \mathcal{O}(n^2)$ candidates.

Including type information this number further reduces to

$$\sum_{i=1}^{k} \binom{t_i}{2} = \sum_{i=1}^{k} t_i(t_i - 1)/2.$$

In the following, the set of typed object transpositions is denoted by $\mathcal{SYMM}$. For the example, we have $|\mathcal{SYMM}| = 45 + 45 + 10 + 105 = 205$.

### 11.6.1 Static Symmetries

We generate a set of object pairs $(o, o') \in \mathcal{SYMM}$, indistinguishable with respect to the set of instantiated operators and the goal specification.

**Definition 19** *(Object Transpositions for Fluents, Variables, and Operators) A transposition of objects $(o, o') \in \mathcal{SYMM}$ applied to a fluent $f = (p\ o_1, \ldots, o_{k(p)}) \in \mathcal{F}$, written as $f[o \leftrightarrow o']$, is defined as $(p\ o'_1, \ldots, o'_{k(p)})$, with $o'_i = o_i$ if $o_i \notin \{o, o'\}$, $o_i = o'$ if $o_i = o$, and $o_i = o$ if $o_i = o'$, $i \in \{1, \ldots, k(p)\}$. Object transpositions $[o \leftrightarrow o']$ applied to a variable $v = (f\ o_1, \ldots, o_{k(f)}) \in \mathcal{V}$ or to an operator $O = (a\ o_1, \ldots, o_{k(a)}) \in \mathcal{O}$ are defined analogously.*

By definition we have

**Lemma 6** *For all $f \in \mathcal{F}$, $v \in \mathcal{V}$, $O \in \mathcal{O}$, and $(o, o') \in \mathcal{SYMM}$: $f[o \leftrightarrow o'] = f[o' \leftrightarrow o]$, $v[o \leftrightarrow o'] = v[o' \leftrightarrow o]$, $O[o \leftrightarrow o'] = O[o' \leftrightarrow o]$, $f[o \leftrightarrow o'][o \leftrightarrow o'] = f$, $v[o \leftrightarrow o'][o \leftrightarrow o'] = v$, and $O[o \leftrightarrow o'][o \leftrightarrow o'] = O$.*

The time complexity for checking $f[o \leftrightarrow o']$ is of order $\mathcal{O}(k(p))$. By precomputing a $\mathcal{O}(|\mathcal{SYMM}| \cdot |\mathcal{F}|)$ sized table containing the index of $f' = f[o \leftrightarrow o']$ for all $(o, o') \in \mathcal{SYMM}$, this time complexity can be reduced to $\mathcal{O}(1)$.

**Definition 20** *(Object Transpositions for States) An* object transposition $[o \leftrightarrow o']$ *applied to state $S = (S_p, S_n) \in \mathcal{S}$ with $S_n = (v_1, \ldots, v_k)$, $k = |\mathcal{V}|$, written as $S[o \leftrightarrow o']$, is equal to $(S_p[o \leftrightarrow o'], S_n[o \leftrightarrow o'])$ with*

$$S_p[o \leftrightarrow o'] = \{f' \in \mathcal{F} \mid f \in S_p \ \wedge \ f' = f[o \leftrightarrow o']\}$$

*and $S_n[o \leftrightarrow o'] = (v'_1, \ldots, v'_k)$ with $v_i = v'_j$ if $\phi^{-1}(i)[o \leftrightarrow o'] = \phi^{-1}(j)$ for $i, j \in \{1, \ldots, k\}$.*

The time complexity to compute $S_n[o \leftrightarrow o']$ is $\mathcal{O}(k)$, since testing $\phi^{-1}(i)[o \leftrightarrow o'] = \phi^{-1}(j)$ is available in time $\mathcal{O}(1)$ by building another $\mathcal{O}(|\mathcal{SYMM}| \cdot |\mathcal{V}|)$ sized precomputed look-up table. We summarize the complexity issues as follows.

**Lemma 7** *The time complexity to compute $S[o \leftrightarrow o']$ for state $S = (S_p, S_n) \in \mathcal{S}$ and $(o, o') \in \mathcal{SYMM}$ is $\mathcal{O}(|S_p| + |\mathcal{V}|)$ using $\mathcal{O}(|\mathcal{SYMM}| \cdot (|\mathcal{F}| + |\mathcal{V}|)$ space.*

**Definition 21** *(Object Transpositions for Domains) A planning problem $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ is symmetric with respect to the object transposition $[o \leftrightarrow o']$, abbreviated as $\mathcal{P}[o \leftrightarrow o']$, if $\mathcal{I}[o \leftrightarrow o'] = \mathcal{I}$ and for all $G \in \mathcal{G}$: $G[o \leftrightarrow o'] \in \mathcal{G}$.*

Applying Lemma 7 for all $(o, o') \in \mathcal{SYMM}$ yields

**Theorem 11** *Assuming a description complexity $\mathcal{O}(|\mathcal{G}_p| + |\mathcal{V}|)$ for the set of goals $\mathcal{G}$, checking whether a planning problem $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ is symmetric with respect to the object transpositions $[o \leftrightarrow o']$, with $(o, o') \in \mathcal{SYMM}$ can be done in time $\mathcal{O}(|\mathcal{SYMM}| \cdot (|\mathcal{G}_p| + |\mathcal{I}_p| + |\mathcal{V}|))$.*

**Lemma 8** *If operator $O$ is applicable in $S$ and $S = S[o \leftrightarrow o']$ then $O[o \leftrightarrow o']$ is applicable in $S$ and*

$$O(S)[o \leftrightarrow o'] = O[o \leftrightarrow o'](S)$$

**Proof:** If $O$ is applicable in $S$ the $O[o \leftarrow o']$ is applicable in $S[o \leftarrow o']$. Since $S = S[o \leftrightarrow o']$, $O[o \leftrightarrow o']$ applicable in $S$, and

$$O[o \leftrightarrow o'](S) = O[o \leftrightarrow o'](S[o \leftrightarrow o']) = O(S)[o \leftrightarrow o'].$$

■

Lemma 8 indicates how symmetry will be used to reduce exploration. If a planning problem with current state $\mathcal{C} \in \mathcal{S}$ is symmetric with respect to the operator transposition $[o \leftrightarrow o']$ then either the application of operator $O \in \mathcal{O}$ or the application of operator $O[o \leftrightarrow o']$ is neglected, significantly reducing the branching factor.

## 11.6.2 Dynamic Symmetries

One problem is that symmetries that are present in the initial state may vanish or reappear during exploration. In the *Desert-Rats* domain, for example, the initial set of supply tanks is indistinguishable so that only one should be loaded into the truck. Once the fuel level of the supply tanks decrease or tanks are transported to another location, formerly existing symmetries are broken. However, when two tanks in one location are emptied they can once more be considered as being symmetric.

In a forward chaining planner goal conditions do not change over time, only the initial state $\mathcal{I}$ transforms to the current state $\mathcal{C}$. Therefore, in a precompiling phase we refine the set $\mathcal{SYMM}$ to

$$\mathcal{SYMM}' := \{(o, o') \in \mathcal{SYMM} \mid \forall\, G \in \mathcal{G} : G[o \leftrightarrow o'] = G\}\,.$$

Usually $|\mathcal{SYMM}'|$ is by far smaller than $|\mathcal{SYMM}|$. For the *Zeno-Travel* instance, the symmetries left in $\mathcal{SYMM}'$ are the ones of the location of `scott` and `ernie`.

**Theorem 12** *Checking whether an induced planning problem $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{C}, \mathcal{G} \rangle$ with current state $\mathcal{C} = (\mathcal{C}_p, \mathcal{C}_n) \in \mathcal{S}$ is symmetric with respect to the object transpositions $[o \leftrightarrow o']$, $(o, o') \in \mathcal{SYMM}'$, can be performed in time $\mathcal{O}(|\mathcal{SYMM}'| \cdot (|\mathcal{C}_p| + |\mathcal{V}|))$.*

Therefore, we can efficiently compute set

$$\mathcal{SYMM}''(\mathcal{C}) := \{(o, o') \in \mathcal{SYMM}' \mid \mathcal{C}[o \leftrightarrow o'] = \mathcal{C}\}$$

of symmetries that are present in the current state. In the initial state of the example problem of *Zeno-Travel* $\mathcal{SYMM}''(\mathcal{C}) = \emptyset$, but once `scott` and `ernie` share the same location in a state $\mathcal{C} \in \mathcal{S}$ this object pair would be included in $\mathcal{SYMM}''(\mathcal{C})$.

By precomputing a $\mathcal{O}(|\mathcal{SYMM}| \cdot |\mathcal{O}|)$ sized table the index of operator $O' = O[o \leftrightarrow o']$ can be determined in time $\mathcal{O}(1)$ for each $(o, o') \in \mathcal{SYMM}'$.

Let $\Gamma(S)$ be the set of operators that are applicable in state $S \in \mathcal{S}$.

**Definition 22** *The* pruning set $\Delta(S, \mathcal{SYMM}''(\mathcal{C})) \subset \Gamma(S)$ *is defined as the set of all operators that have a symmetric operator and that are not of minimal index, i.e.,* $\Delta(S, \mathcal{SYMM}''(\mathcal{C})) =$

$$\{O \in \Gamma(S) \mid \exists\, O' \in \Gamma(S) : \phi(O') > \phi(O) \text{ and } \exists\, (o, o') \in \mathcal{SYMM}''(\mathcal{C}) : O' = O[o \leftrightarrow o']\}\,.$$

*The* symmetry reduction *of* $\Gamma(S, \mathcal{SYMM}''(\mathcal{C})) \subseteq \Gamma(S)$ *with respect to the set* $\mathcal{SYMM}''(\mathcal{C})$ *is defined as* $\Gamma(S, \mathcal{SYMM}''(\mathcal{C})) = \Gamma(S) \setminus \Delta(S, \mathcal{SYMM}''(\mathcal{C}))$.

To shorten notation, in the following we write $\Gamma'(\mathcal{C})$ for $\Gamma(S, \mathcal{SYMM}''(\mathcal{C}))$ and $\Delta'(\mathcal{C})$ for $\Delta(S, \mathcal{SYMM}''(\mathcal{C}))$. Determining $\Gamma'(\mathcal{C})$ can be performed in time $\mathcal{O}(|\Gamma(S)|)$, since finding $O' = O[o \leftrightarrow o']$ and the indices $\phi(O')$ and $\phi(O)$ are all available in constant time.

**Theorem 13** *Reducing the operator set* $\Gamma(\mathcal{C})$ *to* $\Gamma(S, \mathcal{SYMM}''(\mathcal{C}))$ *during the exploration of planning problem* $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ *preserves completeness and sequential optimality for all expanded states* $\mathcal{C}$.

**Proof:** Suppose that for some expanded state $\mathcal{C}$, reducing the operator set $\Gamma(\mathcal{C})$ to $\Gamma'(\mathcal{C})$ during the exploration of planning problem $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ does not preserve completeness and sequential optimality. Furthermore, let $\mathcal{C}$ be the state with this property that is maximal in the exploration order.

Then there is a sequential plan $\pi = \{O_1 \ldots, O_k\}$ in $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{C}, \mathcal{G} \rangle$ with intermediate state sequence $S_0 = \mathcal{C}, \ldots, S_k \subseteq \mathcal{G}$. Obviously, $O_i \in \Gamma(S_{i-1})$, $i \in \{1, \ldots, k\}$. By the choice of $\mathcal{C}$ we have $O_1 \notin \Gamma'(S_0)$. Since $O_1 \notin \Gamma'(S_0)$ but $O_1 \in \Gamma(S_0)$ we have that $O_1 \in \Delta(S_0, \mathcal{SYMM}''(S_0))$. By the definition of the pruning set $\Delta'(S_0)$ there exists $O_1'$, $\phi(O_1') > \phi(O_1)$ and $(o, o') \in \mathcal{SYMM}''(S_0)$ with $O_1' = O_1[o \leftrightarrow o'] \in \Gamma'(S_0)$ that is applicable in $S_0$. By Lemma 8 we have $O_1'(S_0) = S_1[o \leftrightarrow o']$.

Since $\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{C}, \mathcal{G} \rangle = \mathcal{P}[o \leftrightarrow o'] = \langle \mathcal{S}, \mathcal{O}, \mathcal{C}[o \leftrightarrow o'] = \mathcal{C}, \mathcal{G}[o \leftrightarrow o'] = \mathcal{G} \rangle$, we have a sequential plan $O_1[o \leftrightarrow o'], \ldots, O_k[o \leftrightarrow o']$ with state sequence $S_0[o \leftrightarrow o'] = S_0, S_1[o \leftrightarrow o'], \ldots, S_k[o \leftrightarrow o'] = S_k$ that reaches the goal $\mathcal{G}$.

Sequential plan objectives are devised on parameterized predicates and functions, so that any cost function on $O_1[o \leftrightarrow o'], \ldots, O_k[o \leftrightarrow o']$ will be the same as on $O_1, \ldots, O_k$. This contradicts the assumption that reducing the operator set $\Gamma(\mathcal{C})$ to $\Gamma'(\mathcal{C})$ does not preserve completeness and optimality for all $\mathcal{C}$. ■

If the plan objective is defined on instantiated predicates and objects, it can be symmetry breaking and to preserve optimality should be checked as an additional requirement similar to $\mathcal{G}$ and $\mathcal{I}$.

### 11.6.3 Symmetry Reduction in MIPS

The main purpose of the refined implementation in MIPS is to reduce the time for dynamic symmetry detection from $\mathcal{O}(|\mathcal{SYMM}'| \cdot (|\mathcal{C}_p| + |\mathcal{V}|))$ to time $\mathcal{O}(|\mathcal{C}_p| + |\overline{\mathcal{SYMM}'}| \cdot |\mathcal{V}|)$ by loosing some but not all structural properties.

The key observation is that symmetries are also present in fact groups according to their object representatives. Fact groups $G_i \subseteq \mathcal{F}$, $i \in \{1, \ldots, l\}$ implicitly define projections $\mathcal{P}|_i$ of the (propositional) planning space $\mathcal{P}$ by $\mathcal{P}|_i = \langle \mathcal{S}|_i, \mathcal{O}|_i, \mathcal{I}|_i, \mathcal{G}|_i \rangle$, with $\mathcal{S}|_i = G_i$, $\mathcal{I}|_i = \mathcal{I} \cap G_i$, $\mathcal{G}|_i = \bigcup_{G \in \mathcal{G}} G \cap G_i$, and $\mathcal{O}|_i = \{(\beta_a, \beta_b) \in \mathcal{O} \mid (\beta_a \cup \beta_b) \cap \mathcal{S}|_i \neq \emptyset\}$. By construction for all $S \in \mathcal{S}$ we have exactly one fact in each group true, such that $S$ can be partitioned into $\{S_1, \ldots, S_l\}$, with $S_i \in \mathcal{S}|_i$, $i \in \{1, \ldots, l\}$.

Let $R_i \subseteq \mathcal{OBJ}$ be the set of object representatives for group $G_i$. If $S[o \leftrightarrow o'] = S$ then $\mathcal{S}|_i[o \leftrightarrow o'] = \mathcal{S}|_j$ in a group $G_j$ with representative $R_i[o \leftrightarrow o']$. Hence, in MIPS we devise a symmetry relation $\overline{\mathcal{SYMM}}$ not on objects but on fact groups, i.e.

$$\overline{\mathcal{SYMM}} = \{(i, j) \mid 1 \leq i < j \leq l : R_i[o \leftrightarrow o'] = R_j\}.$$

Many objects, e.g. the objects of type `city` in *Zeno-Travel*, were not selected as representatives for a single attribute invariance to build a group. These were neglected in MIPS, since we expect no symmetry on them. This reduces the set of objects $\mathcal{OBJ}$ that MIPS considers to a considerably smaller subset $\mathcal{OBJ}' = \bigcup_{\{1 \leq i \leq l\}} R_i$. In the example problem $|\mathcal{OBJ}| = 7$, and $|\mathcal{OBJ}'| = 4$.

It may also happen that more than one group has a representative $o \in \mathcal{OBJ}'$. However, if all fluent predicates $p$ have arity $k(p) \leq 2$, which is frequently met in the benchmark domains, all $|R_i|$ were equal to one for all $i$, so for all objects we get a finite partitioning into representatives, i.e. $\mathcal{OBJ}' = \dot{\bigcup}_{i \in \{1,\ldots,l\}} R_i$.

MIPS takes this conservative assumption and may leave other symmetries uncaught. It computes $\overline{\mathcal{SYMM}}$ by analyzing the subproblem structures $\mathcal{P}|_i$, $i \in \{1, \ldots, l\}$ instead of $\mathcal{P}$ itself. In case of an object symmetry $[R_i \leftrightarrow R_j]$ the groups $G_i$ and $G_j$ necessarily have to be isomorphic, and we can establish a bijective mapping $\psi : \mathcal{P}|_i \rightarrow \mathcal{P}|_j$ with subcomponents $\psi_{\mathcal{S}} : \mathcal{S}|_i \rightarrow \mathcal{S}|_j$ and $\psi_{\mathcal{O}} : \mathcal{O}|_i \rightarrow \mathcal{O}|_j$.

As above, static symmetries based on non-matching goal predicates were excluded, yielding a refinement $\overline{\mathcal{SYMM}'}$ of $\overline{\mathcal{SYMM}}$. Dynamic symmetries are detected for each expanded state $S$. The current state representation is mapped to the subgraphs $\mathcal{P}|_i$. The list of possibly symmetric groups $\overline{\mathcal{SYMM}'}$ is traversed to select pairs which obey the current instantiation. MIPS marks the groups with larger index as *visited*. This guarantees that operators of at least one group are executed. The complexity of this phase is bounded by $|S_p|$ and by $\overline{\mathcal{SYMM}'}$ and yields a list $\overline{\mathcal{SYMM}''}$.

Testing the propositional part $S_p = (S_1, \ldots, S_l)$ of a state $S$ for all symmetries reduces to test, whether $\psi_{\mathcal{S}}(S_i) = S_j$ for each $(i, j) \in |\overline{\mathcal{SYMM}'}|$ and can be performed in time $\mathcal{O}(|S_p| + |\overline{\mathcal{SYMM}'}|)$. The comparison of variables $v \in \mathcal{V}$ is implemented as described in the previous section such that for the numerical part $S_n$ we check the remaining symmetries, for total time $\mathcal{O}(|S_p| + |\overline{\mathcal{SYMM}'}| \cdot |\mathcal{V}|)$ to fix $\overline{\mathcal{SYMM}''}$ in form of visited markings.

For each expanded state $S$ and each matching operator $O \in \Gamma(S)$ the algorithm checks, whether an applied operator is present in a visited group, in which case it is pruned. The time complexity is in $\mathcal{O}(|\Gamma(S)|)$, since operator group containment can be preprocessed and checked in constant time.

## 11.7 Visualization

For visualization of plans we extended an existing animation system for our purposes.

Vega [175] is implemented as a Client-Server architecture that runs an sorce-code annotated algorithm on server side to be visualized on client side in a Java frontend. The client is used both as the user front-end and the visualization engine. Thus, it allows server and algorithm selection, input of data, running and stopping algorithms, and customization of the visualization.

It can be used to $i)$ manipulate scenes with hierarchically named objects — either in the view or in an object browser that displays the object tree, $ii)$ view algorithm lists at the server and display algorithm information, $iii)$ apply algorithms to selected data in a view, control the algorithm execution using a VCR-like panel or the execution browser, $iv)$ adjust view attributes directly or using the object browser, show several algorithms simultaneously in multiple scenes and open different views for a single scene, and $v)$ load and save single scenes, complete runs, and attribute lists, export scenes in xfig or gif format.

Vega allows on-line and off-line presentations. The main purpose of the server is to make algorithms accessible through TCP/IP. The server is able to receive commands from multiple clients at the same time. It allows the client to choose from the list of available algorithms, to retrieve information about the algorithm, to specify input data, to start it and to receive output data. The server maintains a list of installed algorithms. This list may be changed without the need of stopping and restarting the server.

We have extended Vega with two respects, and call it Vepa for *Visualization of Efficient Planning Algorithms* to emphasize the planning aspect. It can be run as an interactive applet available at `www.informatik.uni-freiburg.de/ mmips/visualization`.

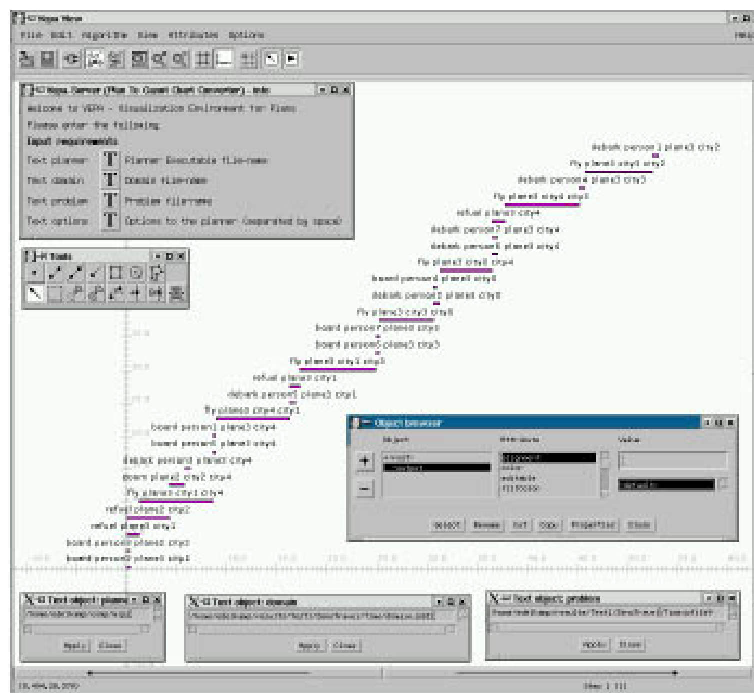

Figure 11.10: Visualization a Plan in Gannt Chart Format.

The first program that we added is *VepaServer* which wraps plan execution and visualizes Gannt Charts of plans, see Figure 11.10. Gannt Charts are a well known representation for schedules in which a horizontal open oblong is drawn against each activity

indicating estimated duration. The tool can be adapted to any planner that writes plans in planning competition format to standard I/O. The user may choose the planner, the domain, and the problem file.

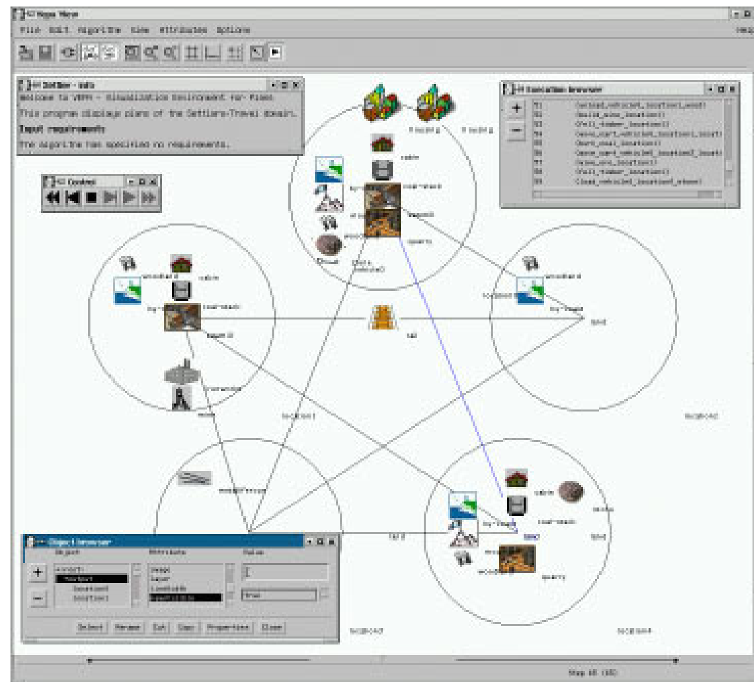

Figure 11.11: Visualization of a Planning Problem Instance of Settlers.

The second program (suite) is *VepaDomain* for domain-dependent visualization of sequential plans. Figure 11.10 shows an example for the *Settlers* domain. *VepaDomain* includes (almost) instance independent visualizations for all competition domains all with less than 100 lines of code. The gifs for the objects were collected with an image web search engine. The planner MIPS writes propositional and numeric state facets and action sequences into a header-file, which in turn is compiled together with generic c-file to visualize the sequential plan sequence.

## 11.8 Related Work

Solving planning problems with numerical preconditions and effects as allowed in Level 2 and Level 3 problems is undecidable in general [170]. However, the structures of the provided benchmark problems are simpler than the general problem class, so that these problems are in fact solvable.

### 11.8.1 Problem Classes and Methods

According to the PDDL-hierarchy we indicate three problem classes:

1. Propositional Planning. STRIPS problems have been tackled with different planning techniques, most notably by SAT-planning, e.g. [204], IP-planning, e.g. [205], CSP-planning, e.g. [307], graph relaxation, e.g. [39], and heuristic search planing,

e.g. [41]. The major quality measurements are the numbers of sequential and parallel steps. ADL generalizations [289] like conditional effects and negative preconditions are more expressive in general, but can usually be resolved during grounding.

2. Numerical Effects. Numerical variables in the effect lists can include time and resources. If numerical effects do not bound integral values, infinite state spaces are likely to be generated. However, by assuming finitely many interesting events the problem class becomes tractable and is effectively dealt by schedulers that usually minimize the *make-span* of concurrent actions.

3. Numerical Preconditions. We distinguish finite and infinite branching problems. With finite branching, execution time of an action is not parameterized, while with infinite branching, an infinite number of actions can be applied. These problems have ever since been confronted to model checking. Some subclasses of infinite branching problems like timed automata exhibit a finite partitioning through a symbolic representation of states [292]. By the technique of shortest-path reduction a unique and reduced normal form can be obtained. We have implemented this constraint network data structure, since this is the main data structure when exploring timed automata as done by the model checker Uppaal [292]. For this to work, all constraints must have the form $x_i - x_j \leq c$ or $x_i \leq c$. For example, the set of constraints $x_4 - x_0 \leq -1$, $x_3 - x_1 \leq 2$, $x_0 - x_1 \leq 1$, $x_5 - x_2 \leq -8$, $x_1 - x_2 \leq 2$, $x_4 - x_3 \leq 3$, $x_0 - x_3 \leq -4$, $x_1 - x_4 \leq 7$, $x_2 - x_5 \leq 10$, and $x_1 - x_5 \leq 5$ has the shortest-path reduction $x_4 - x_0 \leq -1$, $x_3 - x_1 \leq 2$, $x_5 - x_2 \leq -8$, $x_0 - x_3 \leq -4$, $x_1 - x_4 \leq 7$, $x_2 - x_5 \leq 10$, and $x_1 - x_5 \leq 5$. If the constraint set is over-constraint, the algorithm will determine unsolvability, otherwise a feasible solution is returned. The absence of partitioning is current research [349].

Critical path analysis for timed precedence networks is one of the simpler cases for scheduling. We have achieved a simplification by solving the sequential path problem first. Several scheduling techniques apply the presented critical path analyses as a subcomponent [335].

Most previously achieved results in symmetry reduction, e.g. [153], neglect the combinatorial explosion problem and tend to assume that the information on existing symmetries in the domain is supplied by the user. Our approach shares similarities with the approach of [130, 132] in inferring symmetry information automatically, which bases on the TIM inference module [129]. Since no additional information on the current symmetry level in form of matrix is stored, our approach consumes less space per state. Moreover, we can give correctness proofs and efficiency guarantees.

### 11.8.2 Competing Planners

The on-line presentation of IPC-3[4] provides aspects of the input language, domains, results and other resources, e.g links to competing planners and the history of the event. In the following we briefly present the successful approaches at AIPS-2002. In AIPS-1998 most successful planners besides HSP [41] and *Satplan* [204] were *Graphplan* derivates,

[4] http://www.dur.ac.uk/d.p.long/competition.html

e.g. IPP [212] and STAN [243]. In 2000, the field was dominated by the success of heuristic search planning as in FF [181], HSP-2 [163], and in some hybrids, like STAN4 [246], and MIPS. System R [238] used backward regression.

Metric-FF [179] extends FF [181] and is a forward chaining heuristic state space planner. It performed best in the numerical track and was the only system besides MIPS that solved instances to *Settlers*. The main heuristic of relaxed plans bases on the HSP-heuristic [41]. Metric-FF deals with PDDL 2.1 level 2, combined with ADL. The key difference is the definition of the relaxation. In STRIPS, the task is relaxed by ignoring all delete lists. However, numerical constraints are not monotonic: while one constraint (e.g. $x > 2$) might prefer higher values of a variable $x$, another constraint (e.g. $x < 2$) might prefer lower values. Opposed to that, the conditions in the purely logical case all prefer *higher* values of the propositional variables: negative conditions are compiled away as a pre-process, and thus it is always preferable to have more propositional facts true. The observation exploited in Metric-FF is that the same methodology can be applied in the numerical setting, at least in a subset of the language. The task is pre-processed such that all numerical constraints are monotonic, i.e., for any constraint *c*, if *c* is true in a state S then *c* is true in any state $S'$ where, for all variables $x$, $x(S') \geq x(S)$. The relaxation is then simply to ignore all effects that decrease the value of the affected variable, and the relaxed task can be solved in Graphplan-style. To achieve the monotonicity property, one needs, in the numerical constraints and effects, expressions that are monotonic in all variables. In the current implementation, Numerical-FF restricts to linear expressions which obviously have this property.

LPG (Local search for Planning Graphs) [143] is the only planner that was competitive with MIPS at AIPS-2002 in the temporal domains. It bases on local search and planning graphs that handles PDDL 2.1 domains involving numerical quantities and durations. The system can solve both plan generation and plan adaptation problems. The basic search scheme of LPG was inspired by Walksat, an efficient procedure to solve SAT-problems. The search space of LPG consists of *action graphs* [142], particular subgraphs of the planning graph representing partial plans. The search steps are certain graph modifications transforming an action graph into another one. LPG exploits a compact representation of the planning graph to define the search neighborhood and to evaluate its elements using a parametrized function, where the parameters weight different types of inconsistencies in the current partial plan, and are dynamically evaluated during search using discrete Lagrange multipliers. The evaluation function uses some heuristics for estimate the *search cost* and the *execution cost* of achieving a (possibly numeric) precondition. Action durations and numerical quantities (e.g., fuel consumption) are represented in the actions graphs, and are modeled in the evaluation function. In temporal domains, actions are ordered using a *precedence graph* that is maintained during search, and that took into account the mutex relations of the planning graph.

TP4 [164] is in fact a scheduling system based on grounded problem instances. For these cases all formula trees in numerical conditions and assignments reduce to constants. Utilizing admissible heuristics TP4 minimize the plan objective of optimal parallel plan length. Our planner has some distinctive advantages: it handles numerical preconditions, instantiates numerical conditions on the fly and can cope with complex objective functions. Besides input restriction, in the competition, TP4 was somewhat limited by is focus to produce optimal solutions only.

SAPA [83] is a domain-independent time and resource planner that can cope with met-

rics and concurrent actions. It adapts the forward chaining algorithm of [16]. Both planning approaches instantiate actions on the fly and can, therefore, in principle be adapted to at least mixed propositional and numerical planning problems. The search algorithm of SAPA extends partial concurrent plans. It uses a relaxed temporal planning graph for the yet unplanned events for different heuristic evaluation functions. In the competition SAPA was the only system besides MIPS that produced plans for the complex domains, which was the only one it submitted solutions to.

### 11.8.3 Symbolic Model Checking based Planners

In the 2000 competition, two other symbolic planner took part: PropPlan [128], and BDDPlan [182]. Although they were not awarded for performance, they show interesting properties. PropPlan performs symbolic forward breadth first search to explore propositional planning problems with propositions for generalized action preconditions and generalized action effects. It performed well in the full ADL Micsonic-10 elevator domain [211]. ProbPlan is written in the Poly/ML implementation of SML and the standart C-BDD library[5]. BDD-Plan bases on solving the entailment problem in the fluent calculus with BDDs. At that time the authors acknowledged that a concise domain encoding and symbolic heuristic search as found in MIPS provides a large space for improvements.

In the Model-Based Planner, MBP[6], the paradigm of planning as symbolic model checking [145] has been implemented for *non-deterministic planning* domains [63], which classifies in weak, strong, and strong-cyclic planning, with plans that are represented as complete state-action tables. For *partial observable planning*, exploration faces the space of belief states; the power set of the original planning space. Therefore, in contrast to the successor set generation based on action application, observations introduce “And” nodes into the search tree [32]. Since the approach is a hybrid of symbolic representation of belief states and explicit search within the “And”-“Or” search tree, simple heuristic have been applied to guide the search. The need for heuristics that trade information gain for exploration effort is also apparent need in *conformant planning* [31]. Recent work [30] proposes improved heuristic for belief space planning. MBP has not yet participated in a planning competition, but plan to do in 2004.

The UMOP system parses a non-deterministic agent domain language that explicitly defines a controllable system in an uncontrollable environment [196]. The planner also applies BDD refinement techniques such as automated transition function partitioning. New result for the UMOP system extends the setting of weak, strong and strong cyclic planning to adversarial planning, in which the environment actively influences the outcome of actions. In fact, the proposed algorithm joins aspects of both symbolic search and game playing. UMOP has not participated yet in a planning competition.

More recent developments in symbolic exploration are expected to influence automated planning in near future. With SetA*, [197] provide an improved implementation of the symbolic heuristic search algorithm BDDA* [112] and Weighted BDDA* [93]. One major surplus is to maintain a finer granularity of the sets of states in the search horizon kept in a matrix according to matching $g$- and $h$- values. This contrasts the plain bucket representation of the priority queue based on $f$-values. The heuristic function is implicitly

[5] http://www-2.cs.cmu.edu/ modelcheck/bdd.html

[6] http://sra.itc.it/tools/mbp

encoded with value differences of grounded actions. Since sets of states are to be evaluated and some heuristics are state rather than operator dependent it has still to be shown how general this approch is. As above the considered planning benchmarks are seemingly simple for single-state heuristic search exploration [180, 169]. [158] also re-implemented BDDA* and suggest that symbolic search heuristics and exploration algorithms are probably better to be implemented with algebraic decision diagrams (ADDs), as available in Somenzi's CUDD package. Although the authors achieved no improvement to [112] to solve the $(n^2 - 1)$-Puzzle, the established generalization to guide a symbolic version of the LAO* exploration algorithm [157] for *probabilistic* (MDP) planning, results in a remarkable improvement to the state-of-the-art [124].

## 11.9 Conclusions

With the competing planning system MIPS, we have contributed an object-oriented architecture for a forward chaining, heuristic search explicit and symbolic planner that finds plans in finite-branching numerical problems. The planner parses, pre-compiles, solves, and schedules all current benchmark problem instances, including complex ones with duration, resource variables and different objective functions.

Model checking aspects have always been influencing to the development of MIPS, e.g in the static analysis to minimize the state description length, in symbolic exploration and plan extraction, in the dependence relation for PERT schedules according to a given partial order, in bit-state hashing for IDA*, etc. The successes of planning with MIPS were also exported back to model checking, as the development of a heuristic search explicit-state model checker HSF-SPIN [105] indicates.

MIPS instantiates numerical pre- and postconditions on-the-fly and produces optimized parallel plans. Essentially planning with numerical quantities and durative actions is planning with time and resources. The given framework of mixed propositional and numerical planning problems problems and the presented intermediate format can be seen as a normal form for temporal and metric planning.

For temporal planning, MIPS generates sequential (totally ordered) plans and efficiently schedules them with respect to the set of actions and the imposed causal structure, without falling into known NP-hardness traps for optimized partial-ordering of sequentially generated plan. For smaller problems the enumeration approach guarantees optimal solutions. To improve solution quality in approximate enumeration, the (numerical) estimate for the number of operators was substituted by scheduling the relaxed plan in each state.

Other contributions besides the new expressivity were refined static analysis techniques to simplify propositionally grounded representation and to minimize state encoding, automated state-based dynamic symmetry detection, as well as effective hashing and transposition cuts.

In the main part of paper we have analyzed completeness and optimality of different forms of exploration and have given a troughout theoretical treatment of PERT scheduling and symmetry detection, proving correctness results and studying run-time complexities.

# Part IV

# Theorem Proving

# Paper 12

# Directed Automated Theorem Proving

Stefan Edelkamp and Peter Leven
Institut für Informatik
Georges-Köhler-Allee 51
Am Flughafen 17
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

In *Logic for Programming, Artificial Intelligence and Reasoning*, 2002, To appear.

**Abstract**

This paper analyzes the effect of heuristic search algorithms like A* and IDA* to accelerate proof-state based theorem provers.

A functional implementation of possibly weighted A* is proposed that extends Dijkstra's single-source shortest-path algorithm. Efficient implementation issues and possible flaws for both A* and IDA* are discussed in detail.

Initial results with first and higher order logic examples in *Isabelle* indicate that *directed automated theorem proving* is superior to other known general inference mechanisms and that it can enhance other proof techniques like model elimination.

## 12.1 Introduction

Theorem proving is at the computational core of many Artificial Intelligence (AI) systems to draw inferences in logical models of the real world. Proof-state based systems implicitly span large and infinite state-spaces by generating successor proof-states through the application of encoded rules in the theory. Inference even in simpler theories like first-order logic (FOL) is semi-decidable [140], calling for user interaction, language limitation, domain-specific knowledge, or incomplete inference procedures.

Due to the amount of user intervention, *theorem proving* is often contrasted to *model checking* and to *action planning*, where the inference process is claimed to be *push-button*. However, the gap is smaller than expected on the first glance, since model checking stop-watch automata [172], or temporal and numerical planning [170] also face undecidability results and call for the design of enumerating inference procedures. Moreover, additional hand-coded control rules significantly improve run time [17].

This paper addresses *automated theorem proving* (ATP), where proof-finding is encapsulated in form of a general proof-independent search procedure. Semi-automated techniques that provide control knowledge assist exploration in form of *tactics* and *rule subsets*, restricting the range of applicable rules to a manageable one, in turn pruning exploration space.

The paper is structured as follows. First, it introduces heuristic graph search, especially A* and IDA*, and their implementation in a functional programming language. Next it addresses design and properties of different heuristic functions. Initial experiments oppose blind to heuristic search and evaluate the effect of the proposed estimates to accelerate automated proof generation in the *Isabelle* theorem proving system. We indicate flaws in *Isabelle*'s implementation of the *best-first* procedure for the restricted class of finite graphs. Furthermore, we have integrated A* in a model elimination reasoner to solve challenging instances from the TPTP problem library. We relate our results to specified proof methods like resolution and tableaux and to earlier findings in guided exploration. Additionally, recent work in *model checking* and *action planning* certifies the importance of guided traversals in large state spaces indicating possible cross-fertilizations. Finally, we draw conclusions.

## 12.2 Functional Heuristic Search

Theorem proving distinguishes between backward regression (or top-down) and forward progression (or bottom-up) proofs. This paper is restricted to top-down proofs, since generic search algorithms are better suited to this case. Pure and efficient bottom-up procedures require restrictions like Horn clause reasoning or bounding the instantiations of formulae. Bottom-up reasoning lacks goal information and, therefore, does not fit well to directed inference. The usual exploration scheme for top-down proof procedures depth-first search (DFS) blindly explores the state-space and, if successful, often finds proofs in very large depths of a search tree. The depth of the exploration is sometimes thresholded to terminate exploration and to find shorter solutions. Choosing the right bound prior to the search is difficult and successively increasing the depth bound – known as *iterative deepening* – is computationally expensive.

We model (proof-) state space search as an implicit graph traversal. Hence, we assume

the existence of an underlying weighted graph $G = (V, E, w)$, likely to be too large to be fully traversed and, moreover, to be infinite in many cases.

Heuristic search algorithms take additional search information in form of an evaluation function into account that returns a number for each node to describe the desirability of expanding it. When the nodes are ordered so that the one with the best evaluation value is expanded first and if the evaluation function estimates the cost of the cheapest path from the current state to a desired one, the resulting greedy best-first search strategy (BF) often finds solutions quickly. However, it may suffer from the same defects as DFS – it is not optimal and may be stuck in dead-ends or local minima. Breadth-first search (BFS), on the other hand, is complete and optimal but very inefficient.

BFS and BF are the two extremes of the A* algorithm [161]. A* combines the cost of the generating path $g(u)$ and the estimated cost of the cheapest path $h(u)$ to the goal yielding the cost value $f(u) = g(u) + h(u)$ of the cheapest solution through $u$. Weighted A* [287] scales the influence of $g$ and $h$, fixing $f$ as $w_g g + w_h h$. If $w_g = 1 - w_h = 0$ we obtain BF and if $w_h = 1 - w_g = 0$ we get BFS.

Algorithm A* is best understood as a refinement of the single-source shortest path (SSSP) algorithm of Dijkstra [80], in which Bellmann's relaxation $f(v) \leftarrow \min\{f(v), f(u) + w(u, v)\}$ on edge $(u, v)$ is substituted by $f(v) \leftarrow \min\{f(v), f(u) + w(u, v) + h(v) - h(u)\}$. Therefore, A* mimics Dijkstra exploration in a graph, where the edges $(u, v)$ are re-weighted with offset $h(v) - h(u)$. Frequently, the original graph $G$ is uniformly weighted ($w \equiv 1$), so that the cost of a solution equals its length. In contrast to Dijkstra's algorithm, already expanded nodes are placed back into the priority queue representing the search frontier, if edge weights become negative by re-weighting.

Table 12.1 depicts the implementation of A* based on a priority queue data structure *Open* and a dictionary of expanded nodes *Closed*. The node expansion function $\gamma$ generates the successor set of a given node and predicate *goal* identifies the reached goal.

If $h$ is a lower bound, it underestimates the minimal distance to a goal at every node. In this case $h$ is called *admissible*. For admissible heuristics and finite graphs, A* is complete and optimal. In infinite graphs, for A* to be complete and optimal, the costs of each infinite path have to be infinite [287].

The iterative deepening variant of A*, IDA* [214], explores the tree-expansion of the problem graph and tackles the problem of limited memory with increased exploration time. Table 12.1 shows the implementation of IDA* with stack data structure $S$, where $U$ is the threshold for the current iteration and $U'$ is the threshold for the next iteration. For better performance, transposition tables [306] may maintain the heuristic information of the set of explored nodes and reduce the number of re-expansions of nodes due to uncaught duplicates. Since IDA* simulates A* exploration for each possible threshold value, it is also complete and optimal given an admissible estimate.

In functional implementations of heuristic search algorithms, one input parameter is the heuristic function $h$. Furthermore, the successor generation and goal functions, and the initial state are passed to the algorithms as parameters.

Table 12.2 depicts pseudo-code implementations of A* and IDA* in a functional programming language like Scheme, Haskell or ML. For A*, the priority queue *Open* is represented by a list of triples $(g, f, u)$, sorted by ascending $f$-values. We ommit reopening of already expanded nodes on shorter generating paths. If the heuristic function $h$ is *consistent*, i.e. $h(v) - h(u) + 1 \geq 0$ for all $(u, v) \in E$, this is no restriction. In this case, on every path the priority $f$ is monotonic increasing. All re-weighted edges are pos-

```
proc A*(s)
  Open ← {(s, h(s))}; Closed← {}
  while (Open ≠ ∅)
    (u, f(u)) ← deleteMin(Open)
    insert(Closed,u,f(u))
    if (goal(u)) then return u
    for all v in γ(u)
      f'(v) ← f(u) + h(v) − h(u) + 1
      if (search (Open,v))
        f(v) ←retrieve(Open, v)
        if (f'(v) < f(v)) then
          decreaseKey(Open, (v, f'(v))
      elsif (search(Closed, v)) then
        f(v) ←retrieve(Closed, v)
        if (f'(v) < f(v)) then
          delete(Closed, v)
          insert(Open, (v, f'(v))
      else insert(Open, (v, f'(v))
```

```
proc IDA*(s)
  Push(S, s, h(s))
  U ← U' ← h(s)
  while (U' ≠ ∞)
    U ← U'
    U' ← ∞
    while (S ≠ ∅)
      (u, f(u)) ← Pop(S)
      if (goal(u)) then return u
      for all v in γ(u)
        f'(v) ← f(u) + h(v) − h(u) + 1
        if (f'(v) > U) then
          if (f'(v) < U') then
            U' ← f'(v)
        else
          Push(S, v, f'(v))
```

Table 12.1: Imperative implementation of A* (left) and IDA* (right).

itive, and the correctness argument of Dijkstra's algorithm applies. All extracted nodes will have correct $f$-values. The implementation of A* in Table 12.2 uses simultanuous recursion, while IDA* destincts its main loop from the subloops by explicit maintenance of the stack content.

In difference to the imperative implementation, in functional A* *insert* implements dictionary updates within the set of horizon nodes. If the state is already contained in the priority queue, no insertion takes place, thus avoiding duplicates within the queue. However, since the *Closed* list of already expanded states is not modeled, even on finite graphs the functional BF derivate, i.e. $f(v) = (g, h(v), v)$ is no longer complete. An example for this anomaly is given in Fig. 12.1.

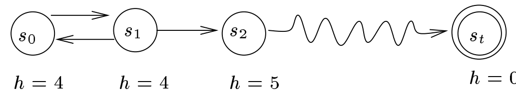


Figure 12.1: Anomaly in which BF without *Closed* list is trapped in an infinite loop.

Starting with state $s_0$, BF will oscillate between $s_0$ and $s_1$. In contrast, for finite state spaces, A* even without duplicate elimination preserves completeness according to the following observation. The total cost of a cycle in the graph is invariant to re-weighting. Let $C = (v_0, \ldots, v_k = v_0)$ be a cycle of length $k$, $w$ the edge weight before, and $w'$ the edge weight after re-weighting. Then

$$w'(C) = \sum_{i=1}^{k} w(v_i, v_{i-1}) + h(v_i) - h(v_{i-1}) = \sum_{i=1}^{k} w(v_i, v_{i-1}) = w(C).$$

```
func A* (s, goal, h, γ) =
  let func relax(succs, t, g) =
    let func f(v) = (g, g + h(v), v);
        l ← (filter goal succs)
    in if (l ≠ []) then l else
      Open(foldr (insert,(map f succs), t))
    end
  and
    func Open [] = [] |
    Open ((g, f, u) :: t) =
      relax (γ(u), t, g + 1)
  in
    relax (γ(s), [ ], 0)
  end
```

```
func IDA* (s, goal, h, γ) =
  let func depth (U, U', []) =
    depth (U', ∞, [(h(s), γ(s))]) |
  depth (U, U', (f, []) :: t) = depth (U, U', t)|
  depth (U, U', (f, succs) :: t) =
    if (f > U) then depth (U, min(U', f), t)
    else let v ← hd succs; succs' ← tl succs
        l ← (filter goal succs)
    in if (l ≠ []) then l else
        depth (U, U', (f + h(v) − h(u) + 1,
          γ(v)) :: (f, succs') :: t)
    end
  in
    depth (0, h(s), [])
  end
```

Table 12.2: Functional implementation of A* and IDA*. Keywords and function declarations are set in bold and variables, function invocations are set in italics.

Therefore, infinite paths have infinite costs. This forces $f$ to exceed any given bound and to eventually generate a final proof-state.

Global expanded node maintenance in *Closed* is integrated in the pseudo codes of Table 12.2 as follows: set *Closed* is supplied as an additional parameter: in A* to the *relax* function, and in IDA* to the *depth* function. In A*, *Closed* is initialized to the empty list at the very beginning, while in IDA*, *Closed* is emptied in each iteration. Instead of (*map* $f$ *succs*) visited states are first eliminated by (*map* $f$ *eliminate*(*Closed*, *succs*)).

The dictionary for *Closed* can be implemented through lists, balanced trees, or low level hash tables. A sorted list implementation for the priority queue *Open* is not time optimal. Nevertheless, more efficient implementations of priority queues like Fibonacci Heaps [134] are challenging. A refined purely functional implementation of a priority queue is provided in [46].

## 12.3 Heuristics for Automated Theorem Proving

Standard TP procedures draw inferences on a set of clauses $\Gamma \to \Delta$, with $\Gamma$ and $\Delta$ as multisets of atoms $\Gamma = \{A_1, \ldots, A_k\}$ and $\Delta = \{B_1, \ldots, B_l\}$ for $k, l \geq 0$. The antecedent $\Gamma$ represents negative literals where the succedent $\Delta$ represents positive literals. When abbreviating $\{\} \to A$ by $A$ and $A \to \{\}$ by $\neg A$, $\Gamma \to \Delta$ can be rewritten as $\neg A_1 \vee \ldots \vee \neg A_k \vee B_1 \vee \ldots \vee B_l$.

The main inference rule[1] is resolution, formally denoted by

$$\frac{\Gamma_1 \to A, \Delta_1 \qquad B, \Gamma_2 \to \Delta_2}{\Gamma_1\tau, \Gamma_2\tau \to \Delta_1\tau, \Delta_2\tau}$$

deriving the conclusion (bottom) from given premises (top), with $\tau$ being the most general

[1] We neglect factoring of a clause.

unifier (mgu) of $A$ and $B$. A top-down proof creates a proof tree, where the node label of each interior node corresponds to the conclusion, and the node labels of its children correspond to the premises of an inference step. Leaves of the proof tree are either axioms or instances of proven theorems.

A proof state represents the outer fragment of a proof tree: the top-node, representing the goal and all leaves, representing the subgoals of the proof state. All proven leaves can be discharged, because they are not needed for further considerations. If all subgoals have been solved, the proof is successful.

One estimate for the remaining distance to the goal is the number of internal nodes of the current proof-state. An illustrative competitor is the string length of a proof state representation.

**Theorem 14** *For the* internal node heuristic *and the* string length heuristic *the number of proof states with fixed heuristic value $k$ is finite.*

**Proof:** The number of trees with $k$ internal nodes and the number of strings with length $k$ are both finite. ■

Theorem 14 is the basis for the design of guided search algorithms with guaranteed progress. At the first glance, heuristic search according to the representation size of a theorem seems not to be a good choice, since it exploits very poor knowledge to prove difficult theorems. Take Fermats theorem $a^n + b^n = c^n$, $n \geq 3$ as an illustrative example. But as we will see in the experiments, even these vague parts of information speed up computation by magnitudes.

The last heuristic we apply is the number of open subgoals in the current proof state. This heuristic is the only one, which is admissible. Recall that all consistent estimates are admissible.

**Theorem 15** *The* open subgoal heuristic *is consistent.*

**Proof:** We have to show that for node $u$ and successor $v$, $h(v) + 1 \geq h(u)$. This, however, is obvious, since the number of unsatisfied subgoals can not decrease by more than 1. ■

In contrast to the *internal node heuristic* and the *string length heuristic* by the limited range of information, the *open subgoal heuristic* can have infinite plateaus of states with the same estimate value. In this case, BF often fails to terminate. For regular state spaces, even weaker heuristics often yield fast solutions in BF. Hence, studying heuristics also yield information on the problem structure and ATP systems characteristics.

## 12.4 Isabelle

There are different state-of-the art generic higher-order logic (HOL) theorem proving systems to which directed search appears applicable, e.g. HOL [150], PVS [281], and COQ [21]. We chose the ML proof-state system *Isabelle* developed at Cambridge University and TU Munich [279] for our experiments.

Isabelle has a rising application focus as projects on compositional reasoning about concurrent programs and on verifying eCommerce protocols show. We worked with the

current Isabelle-2002[2], which is distributed with a wide range object logics, like higher-order logic, classical and intuitionistic first-order logic, set theory (ZF), Horn logic for Prolog programming, just to name a few. There are many specialized logics and Isabelle allows the user to specify their own object logics.

Isabelle is an interactive and tactical theorem prover, supporting forward and backward proofs. In a forward proof axioms and already proven theorems are combined to gain new theorems. In a backward proof, one starts with the theorem to prove, which is step by step reduced to new subgoals. With a tactic, basic inference steps are combined to larger case-sensitive and proof-searching rules using axioms, memorized theorems or assumptions.

For increasing performance in some basic object logics, tableau theorem provers have been integrated into Isabelle, but their inference is not generic for all object logics. The inference process is hidden in the *auto/blast* tactic.

A recent self-contained introduction to interactive proof in Isabelle is [279]. Isabelle is generic, i.e., logics are not hard-wired, but formulated in Isabelle's own meta logic; a fragment of intuitionistic higher order logic.

The only documented heuristic search algorithm in *Isabelle* is BF. As said, this greedy strategy, which always expands the state with minimal evaluation function value, is much attracted by local minima and is not optimal. Even worse, the implementation of BF in Isabelle[3] is not complete on finite graphs. It can be trapped by the anomaly described in Fig. 12.1. In contrast, DFS and BFS are complete. DFS uses global memory to store already proven subgoals and BFS omits all pruning of duplicate states.

Norbert Völker implemented an unpublished version of A* in *Isabelle*. He used a priority queue representation based on linear lists and also omits the *Closed* list. Völker [346] denoted that the initial set of experiments led to no substantial success and that, up to his knowledge, no researcher except him has looked closer to that code.

The implementation of IDA* in Isabelle is specialized. First, it applies the number of subgoals as a fixed heuristic. Then it adjusts the search thresholds within a larger range to allow accelerations for smaller values, e.g. $U$ is initialized to 5 and increased by the minimum of 10 and the value of the best node exceeding the current fringe by twice the heuristic estimate. Moreover, Isabelle's implementation performs *rigid ancestor pruning*, a form of a branching cut for the case a node $v$ with predecessor $u$ is reached with $h(v) - h(u) < 0$ and $f(v) + h(v) \geq U$, for $(u, v) \in E$. Rigid ancestor pruning is related to dynamic transposition table updates, but specialized to the case that the open subgoal heuristic is chosen.

## 12.5 Experiments

In our first implementation of A*, $I_1$ for short, we extended Völker's code to allow arbitrary weighting of the heuristic estimate and the generating path length. This immediately yields BF and BFS. Note that DFS cannot be modeled by a different cost function. In the second implementation of A*, $I_2$ for short, we re-implemented A* from scratch using the internal heap priority queue representation [280]. In contrast to Völker's approach, we

[2]available at `www.cl.cam.ac.uk/Research/HVG/Isabelle`

[3]all top-level inference procedures in *Isabelle* are specified in the file `Proof/search`

| Name | Inference Rule |
|---|---|
| disjI1 | $?P \Rightarrow ?P\|?Q$ |
| disjI2 | $?Q \Rightarrow ?P\|?Q$ |
| disjE | $[\|?P\|?Q; ?P \Rightarrow ?R; ?Q \Rightarrow ?R\|] \Rightarrow ?R$ |
| impI | $(?P \Rightarrow ?Q) \Rightarrow ?P \rightarrow ?Q$ |
| conjunct1 | $?P\&?Q \Rightarrow ?P$ |
| allI | $(!!x.?P(x)) \Rightarrow \forall x.?P(x)$ |
| exE | $[\|\exists x.?P(x); !!x.?P(x) \Rightarrow ?R\|] \Rightarrow ?R$ |
| mp | $[\|?P \rightarrow ?Q; ?P\|] \Rightarrow ?Q$ |
| spec | $\forall\, x.?Px \Rightarrow ?P?x$ |

Table 12.3: Some FOL inference rules in Isabelle.

avoid maintaining $g$-values and compute $f'(v) = f(u) - h(u) + h(v) + 1$. The third implementation $I_3$ additionally maintains the set *Closed* in form of a balanced tree dictionary avoiding to be trapped by the anomaly described above. The specialized IDA* algorithm in Isabelle is referred to as ID+$(C, R, h_3)$, $C$ for constant initial and subsequent offsets, $R$ for rigid ancestor pruning and $h_3$ for the subgoal heuristic, while our generic implementation is denoted by ID(A*). Heuristic $h_1$ refers to the interior node size of the theorem (`size_of_thm`), $h_2$ to its string representation length (`size o string_of_thm`), and $h_3$ to the number of open subgoals (`nprems_of`).

In the first experiment we chose the tautology $P \,\&\, Q \mid R \;\rightarrow\; P \mid R$ in FOL taken from [279]. Each node corresponds to an Isabelle proof state. The successor nodes are generated by applying the following tactic

```
val app_tac = ((assume_tac 1)
  APPEND (resolve_tac [disjI1,disjI2,disjE,impI,conjunct1] 1));
```

The combined tactic is always applied to the first subgoal of a state and the possible outcomes are collected. Other orderings are feasible but have not been considered so far. Additionally, we tried to solve the subgoal by assumption. Table 12.3 depicts a selected subset of FOL inference rules in Isabelle notation; where "$\Rightarrow$" denotes meta logic implication, while "$\rightarrow$" denotes object level implication, and question mark prefixed characters are meta logic variables.

Table 12.4 lists our results for this problem. All CPU results were computed on a Sun Ultra Workstation with 248 MHz and 1.5 GB memory. Time is given in seconds, $\#n$ denotes the number of expanded nodes (A*) and total number of generated nodes (ID(A*)); *t.o* means *time out* and *n.d* denotes *not defined*.

First of all, DFS is not competitive, since it fails to find a proof. Because of this bad performance, we dropped DFS from further considerations. BF is also not a very good choice for this problem. The approach generates very long proofs and performs much work in deeper levels of the search tree. BFS finds a proof, but neither Isabelle's implementation nor $I_1$ are efficient. The reason is that the *Isabelle*'s implementation of BFS does not track duplicates in the search. While $I_2$ memorizes duplicates, the list-like priority queue data structure is very slow. Only the BFS derivate of $I_2$ is competitive. A* is fast in all three implementations. For all heuristics, A* performs better than BF. With $h_3$ we achieved results improving Isabelle's top time performance by two orders of

| | Isabelle | | $I_1$ | | $I_2$ | | $I_3$ | |
|---|---|---|---|---|---|---|---|---|
| | time | $\#n$ | time | $\#n$ | time | $\#n$ | time | $\#n$ |
| DFS | t.o | t.o | n.d | n.d | n.d | n.d | n.d | n.d |
| BFS | 31s | 14,587 | 443s | 9,619 | 6s | 3,200 | 14s | 3,196 |
| A*+$h_1$ | n.d. | n.d. | 3.2s | 2,299 | 1.7s | 2,174 | 1.0s | 596 |
| BF+$h_1$ | 117s | 51,231 | t.o | t.o | 108s | 51,231 | 23s | 7,006 |
| A*+$h_2$ | n.d. | n.d. | 11s | 1,492 | 12s | 1,456 | 4.7s | 470 |
| BF+$h_2$ | 17s | 2,449 | 20s | 2,550 | 21s | 2,449 | 6.2s | 709 |
| A*+$h_3$ | n.d. | n.d. | 0.2s | 231 | 0.2s | 203 | 0.2s | 195 |
| BF+$h_3$ | t.o | t.o | t.o | t.o | t.o | t.o | t.o | t.o |

| | time | $\#n$ |
|---|---|---|
| ID+$(C, R, h_3)$ | 0.4s | 3,555 |
| ID | 563.2s | 72,305 |
| IDA*+$h_1$ | 18.4s | 8,549 |
| IDA*+$h_2$ | 54.4s | 12,265 |
| IDA*+$h_3$ | 11.0s | 3,249 |

Table 12.4: Results in proving $P\&Q|R \rightarrow P|R$ .

magnitudes. Heuristic $h_3$ is consistent, so that A* simplifies to Dijkstra's algorithm on positive weighted graphs. BF with $h_3$ fails, since it introduces an infinite number of new variables in rule `conjunct1`. The implementation of IDA* also shows the effectiveness of heuristic search. Moreover, node expansions are more expensive in the new implementation, indicating possible code tuning, e.g. by refined transposition tables with smaller initialization time.

Our next example $(\forall x.\ P(x) \rightarrow Q) \rightarrow (\exists x.\ P(x)) \rightarrow Q$ is also taken from the introductory book for Isabelle [279]. The set of tactics that we have applied is

```
val app_tac = ((assume_tac 1)
  APPEND (resolve_tac [disjI1,disjI2,disjE,impI,allI,conjunct1] 1)
  APPEND (eresolve_tac [exE,mp] 1) APPEND (dresolve_tac [spec] 1));
```

The rules are distinguished by structural properties. For example, *elimination rules* consume assumptions in premises, yielding one-step resolutions, instantiations, and deletions of meta-assumptions for many cases. Therefore, the tactic `eresolve`, a specialization of resolve, can be used for those rules, e.g. for `exE`. The tactical `APPEND` concatenates lists of tactic functions to a new one, i.e., generating the list of all successful tactic applications for a given subgoal.

Even though the second example looks more complicated, proving is easier as our results in Table 12.5 show. The interpretation of the results is limited, since most runs finished within a second. As in the above example, evaluating the string length in $h_2$ decreases the performance by about one magnitude. Moreover, BF with $h_3$ still poses a problem for exploration, while A* performs well in all cases. The new iterative deepening implementation beats the specialized one in the number of generated nodes but not in CPU time.

Our HOL example is *Cantor's Theorem*

$$\forall f :: \alpha \Rightarrow \alpha \Rightarrow bool.\ \exists S :: \alpha \Rightarrow bool.\ \forall x :: \alpha . f\ x \neq S$$

| | Isabelle | | $I_1$ | | $I_2$ | | $I_3$ | |
|---|---|---|---|---|---|---|---|---|
| | time | $\#n$ | time | $\#n$ | time | $\#n$ | time | $\#n$ |
| BFS | 2.23s | 1,241 | 3.60s | 927 | 0.40s | 238 | 0.71s | 238 |
| A*+$h_1$ | n.d. | n.d. | 0.02s | 29 | 0.02s | 30 | 0.02s | 26 |
| BF+$h_1$ | 0.05s | 46 | 0.02s | 47 | 0.03s | 46 | 0.04s | 42 |
| A*+$h_2$ | n.d. | n.d. | 0.23s | 30 | 0.27s | 30 | 0.24s | 26 |
| BF+$h_2$ | 0.50s | 46 | 0.24s | 34 | 0.25s | 34 | 0.23s | 30 |
| A*+$h_3$ | n.d. | n.d. | 0.05s | 68 | 0.03s | 41 | 0.04s | 41 |
| BF+$h_3$ | 0.03s | 46 | t.o | t.o | 0.03s | 46 | 0.04s | 42 |

| | time | $\#n$ |
|---|---|---|
| ID+$(C, R, h_3)$ | 0.1s | 578 |
| ID | 66.31s | 9,074 |
| IDA*+$h_1$ | 0.1s | 151 |
| IDA*+$h_2$ | 0.2s | 70 |
| IDA*+$h_3$ | 1.3s | 520 |

Table 12.5: Results in proving $(\forall x.P(x) \rightarrow Q) \rightarrow (\exists x.P(x)) \rightarrow Q$.

stating that every set has more subsets than it has elements. It is taken from the manual *Isabelle's Logics:HOL* and refers to the context of set theory. The set of inference rules for this case (cf. Table 12.6) is

```
val app_tac = ((assume_tac 1)  APPEND (contr_tac 1)
  APPEND (resolve_tac [notI] 1) APPEND (swap_res_tac [CollectI] 1)
  APPEND (eresolve_tac [rangeE,equalityCE] 1)
  APPEND (dresolve_tac [CollectD] 1));
```

Despite that the results in Table 12.7 show that on this suite of tactics the theorem is even simpler than the one above, it highlights the generality of the directed search approach.

To harden the exploration task with a larger branching factor we integrated some additional[4] HOL rules to the `eresolve` tactic (cf. Table 12.6):

```
[False_neq_True,allE,all_dupE,conjE,contrapos_nn,ex1E,fun_cong,
 iffD2,major,minor,mp,notE,spec,ssubst,subst,sym,theI]
```

The results are shown in Table 12.8. While for A* and BF little differences can be observed, BFS and ID have severe exploration problems. The reason is an exceeded unification bound for higher order variables. Heuristic $h_2$ has an advantage, because full information of the proof states including higher order variables keeps the exploration efforts small.

The last set of examples is devoted to more complicated proofs and the integration of our A* implementation in a model-elimination reasoner. We took the MESON implementation of Lawrence Paulson in the Isabelle system as a wellcome original source (`meson.ML`). We randomly chose 3 hard examples from the TPTP library suite (in Isabelle), namely LDA003, MSC006, and PUZ025. Original MESON applies the algorithm ID$(C, R, h_3)$ as a subroutine. We changed the specialized iterative deepening variant with A* in implementation $I_3$ and heuristic $h_1$-$h_3$. Table 12.9 shows that A* yields a significant

[4] We established the set by a simple UNIX fgrep of `eresolve` in the `HOL/ex` directory.

| Name | Inference Rule |
|---|---|
| rangeI | $?f?x : range?f$ |
| notI | $(?P \Rightarrow False) \Rightarrow\ ?P$ |
| CollectI | $?P?a \Rightarrow ?a : Collect?P$ |
| rangeE | $[\vert ?b : range?f; !!x.?b = ?fx \Rightarrow ?P \vert] \Rightarrow ?P$ |
| equalityCE | $[\vert ?A = ?B; [\vert ?c : ?A; ?c : ?B \vert] \Rightarrow ?P; [\vert ?c\ : ?A; ?c\ : ?B \vert] \Rightarrow ?P \vert] \Rightarrow ?P$ |
| CollectD | $?a : Collect?P \Rightarrow ?P?a$ |
| False_neq_True | False $=$ True $\Rightarrow ?P$ |
| allE | $[\vert \forall\, x.?Px; ?P?x \Rightarrow ?R \vert] \Rightarrow ?R$ |
| all_dupE | $[\vert \forall\, x.?Px; [\vert ?P?x; \forall\, x.?Px \vert] \Rightarrow ?R \vert] \Rightarrow ?R$ |
| conjE | $[\vert ?P\&?Q; [\vert ?P; ?Q \vert] \Rightarrow ?R \vert] \Rightarrow ?R$ |
| contrapos_nn | $[\vert\ ?Q; ?P \Rightarrow ?Q \vert] \Rightarrow\ ?P$ |
| ex1E | $[\vert \exists x.?Px; !!x.[\vert ?Px; \forall\, y.?Py \rightarrow y = x \vert] \Rightarrow ?R \vert] \Rightarrow ?R$ |
| fun_cong | $?f = ?g \Rightarrow ?f?x = ?g?x$ |
| iffD2 | $[\vert ?P = ?Q; ?Q \vert] \Rightarrow ?P$ |
| major | $\exists\, x.P'x[.]$ |
| minor | $P \Rightarrow\ P[.]$ |
| notE | $[\vert\ ?P; ?P \vert] \Rightarrow ?R$ |
| ssubst | $[\vert ?t = ?s; ?P?s \vert] \Rightarrow ?P?t$ |
| subst | $[\vert ?s = ?t; ?P?s \vert] \Rightarrow ?P?t$ |
| sym | $?s = ?t \Rightarrow ?t = ?s$ |
| theI | $[\vert ?P?a; !!x.?Px \Rightarrow x = ?a \vert] \Rightarrow ?P(The?P)$ |

Table 12.6: Some HOL inference rules in Isabelle.

improvement in the first two examples, while its efficiency falls off in the last one. In all terminating cases, A* yields a smaller number of inference steps. Although the number of generated nodes is always larger than the number of expanded nodes, the difference in values of $\#n$ is still large.

## 12.6 Related Work

The combination of classical heuristic search with general proof-state-based ATP can seldom be found in literature. The closest match is probably the project *learning search heuristics for theorem proving* of the DFG research programme *Deduktion*. However, we are not aware of any work that applied A* to ATP.

The basic meta inference rule, which is used in all Isabelle examples is resolution. More specialized resolution calculi restrict inference by global restrictions (e.g. ordered resolution, hyperresolution) or local conditions (e.g. clausal selection function). With this respect, the use of heuristic evaluation functions on proof states also guides the inference process for proof construction. Heuristic can determine global as well as local preferences in order to reach the goal.

Model elimination [248] is known to be space efficient. The refutation complete calculus for first order logic is the basic inference procedure of some modern theorem provers [332, 235]. Unsatisfiability of a clause set is shown by expanding branches of literal labeled trees. The initial tree consists of one node labeled with the empty clause. In each expansion step, a literal $L$ on a branch is unified with a clause, such that the unified clause

| | Isabelle | | $I_1$ | | $I_2$ | | $I_3$ | |
|---|---|---|---|---|---|---|---|---|
| | time | $\#n$ | time | $\#n$ | time | $\#n$ | time | $\#n$ |
| BFS | 0.1s | 40 | 0.1s | 40 | 0.0s | 14 | 0.0s | 14 |
| A*+$h_1$ | n.d | n.d | 0.0s | 10 | 0.0s | 10 | 0.0s | 10 |
| BF+$h_1$ | 0.0s | 10 | 0.0s | 10 | 0.0s | 10 | 0.0s | 10 |
| A*+$h_2$ | n.d | n.d | 0.2s | 29 | 0.2s | 30 | 0.0s | 8 |
| BF+$h_2$ | 0.4s | 46 | 0.2s | 47 | 0.2s | 46 | 0.0s | 8 |
| A*+$h_3$ | n.d | n.d | 0.2s | 29 | 0.2s | 30 | 0.0s | 11 |
| BF+$h_3$ | 0.4s | 46 | 0.2s | 47 | 0.2s | 46 | 0.0s | 10 |

| | time | $\#n$ |
|---|---|---|
| ID+$(C, R, h_3)$ | 0.1s | 30 |
| ID | 0.32s | 111 |
| IDA*+$h_1$ | 0.05s | 44 |
| IDA*+$h_2$ | 0.1s | 44 |
| IDA*+$h_3$ | 0.05s | 67 |

Table 12.7: Results in proving Cantor's theorem.

contains a contrapositive of literal $L$. For every literal of the unified clause new branches and literal labels are added to the end of the considered branch. Branches containing contrapositive literals are closed. A proof is a tree with no open branch.

Model elimination is very space efficient, because on every branch every literal and its contrapositive occur at most once and it is not necessary to extend branches in parallel. Additional inference rule (reduction, contraction) reduce the tree. The context information in each expansion step is the multiset of literals on branches. Nevertheless, the disadvantage is that backtracking is necessary (in the first-order case). Directed theorem proving can be used, to keep the number of inferences small by memoizing proof states.

Paulson uses sophisticated iterative deepening to restrict inferences in the Isabelle's MESON implementation [286]. He denotes, that the MESON procedure crushes `blast_tac` on harder first-order challenge problems. But the tableau-based `blast_tac` is not restricted to pure first-order problems.

Harrison [160] shows on a huge set of hard-provable examples from the TPTP library how different search space restrictions of model elimination, like best-first search, depth-bounded iterative deepening, and inference-bounded iterative deepening can be used to solve efficiently many of the problems. In most of the successful cases best-first strategies perform best.

In general tableaux proof procedures there is no need for backtracking. Therefore proof-state based memoization and guidance is not immediate for semantic tableaux.

Model checking (MC) [65] validates the truth of a temporal property $\phi$ within a given model $M$, abbreviated by $M \models \phi$. It applies to both software and hardware verification. In software – especially communication protocol – validation, the model is either directly specified or transformed into a set of communicating finite state automata. The search for counter-examples for the subclass of safety-properties is modeled as a search for a path in the implicitly spanned state-space. The shorter the error trail, the easier for the designer to trace the error. This objective matches with ATP, where shorter proofs are usually the better ones.

MC and ATP have much more in common. It is not that all deductions are symbolic and that only MC explorations use binary decision diagrams. Also the application areas

| | Isabelle | | $I_1$ | | $I_2$ | | $I_3$ | |
|---|---|---|---|---|---|---|---|---|
| | time | $\#n$ | time | $\#n$ | time | $\#n$ | time | $\#n$ |
| BFS | t.o | t.o | t.o | t.o | 19.2s | 42 | 19.4s | 42 |
| A*+$h_1$ | n.d | n.d | 0.1s | 69 | 0.1s | 59 | 0.2s | 51 |
| BF+$h_1$ | 0.0s | 69 | 0.0s | 69 | 0.0s | 51 | 0.1s | 51 |
| A*+$h_2$ | n.d | n.d | 0.1s | 11 | 0.1s | 11 | 0.1s | 11 |
| BF+$h_2$ | 0.1s | 11 | 0.1s | 11 | 0.1s | 11 | 0.1s | 11 |
| A*+$h_3$ | n.d | n.d | 0.0s | 27 | 0.1s | 26 | 0.0 | 26 |
| BF+$h_3$ | 0.0s | 69 | 0.0s | 25 | 0.0s | 51 | 0.1s | 51 |

| | time | $\#n$ |
|---|---|---|
| ID+$(C, R, h_3)$ | 0.02s | 53 |
| ID | t.o | t.o |
| IDA*+$h_1$ | 0.08s | 70 |
| IDA*+$h_2$ | 0.15s | 70 |
| IDA*+$h_3$ | 22.0s | 138 |

Table 12.8: Extended results in proving Cantor's theorem.

| | LDA003 | | MSC006 | | PUZ025 | |
|---|---|---|---|---|---|---|
| | time | $\#n$ | time | $\#n$ | time | $\#n$ |
| MESON+ID$(C, R, h_3)$ | 297.0s | >119,585 | 17.6s | >19,116 | 75.1s | >121,806 |
| MESON+A*+$h_1$ | 44.1s | 4,330 | 2.05s | 479 | 335.6s | 8,822 |
| MESON+A*+$h_2$ | 256.7s | 4,772 | 13.17s | 394 | 1,764s | 10,663 |
| MESON+A*+$h_3$ | t.o | t.o | t.o | t.o | t.o | t.o |

Table 12.9: Results of proving 3 hard examples by original Meson procedure and A* with Meson

like the verification of a security protocols often match. On the *First World Congress on Formal Methods* Amir Pnueli gave an invited talk on the differences in ATP and MC. In his opinion deduction uses a more expressive language leading to succinct representations of parameterized systems. It is mainly based on induction and requires (at least some) user ingenuity.

Explicit [109] and symbolic heuristic search [303] have a rising influence to MC. An example for a directed explicit state model checker is HSF-SPIN[5] that extends the SPIN verification tool with directed search algorithms like A*, IDA* and BF and has led to drastic reductions in the number of expanded nodes. It separates heuristic that accelerate error-finding from heuristics that improve error-trails, and is compatible with bit-state hashing and partial order reduction.

Action planning (AP) [6] searches for a plan as a sequence of actions that transforms the initial state into one of the goal states. The input divides into a domain and problem specification, usually given in PDDL syntax.

Most performant domain-independent planners include some form of heuristic search. Currently, the best estimates are the relaxed planning heuristic [177] and the pattern database heuristic [95]. In propositional planning due to the highly regular structure of the domains, *enforced hill climbing* (EHC) seems to have a slight advantage to A*. Unfortunately, EHC commits decision to successor nodes, so that it can be trapped in directed

[5]See `http://www.informatik.uni-freiburg.de/ edelkamp/software`

search spaces.

## 12.7 Conclusions and Future Work

Based on the successes in AP and MC the paper re-initiates the research of directed exploration in ATP. Although the basic idea of guided exploration is indeed not new to the TP community, the development of guidance in form of heuristic search procedures has not been elaborated so far.

With the DATP paradigm we confront greedy exploration schemes, like DFS and BF, with more conservative algorithms, like BFS and A*. The lessons to be learned from our experiments are that since A* with efficient priority queue data structure is better than any other search algorithm in most cases, it should be integrated as a standard option to many automated theorem provers like *Isabelle*. Moreover, at least the implementation of BF in *Isabelle* has to be thought of. Isabelle's implementation of IDA* is specialized and not accessible for different heuristics, so that we have provided a generic solution.

The results in the paper present initial findings on a set of benchmark theorems taken from introductory texts for interactive proving and a few hard problems from the TPTP library. We generalized the set of possible inference steps in form of a universal successor generation function. In our HOL example we widened the branching by including additional proof tactics. Our treatment pinpoints the generality and potential of the approach.

Work on suitable polynomial abstractions to yield refined heuristics for the ATP search process is a major challenge for future research, likely to be initiated along the following ideas.

For efficient ATP, grounded representations are essential, since many decision problems can be reduced to the *ground entailment problem*, i.e., to determine the truth of a ground formula for a given theory. Given a set of ground Horn clauses $N$ and a ground clause $C$, the entailment problem $N \models C$ is decidable in linear time [86]. We aim to combine such object-logic independent decision procedures for local clause sets [22] with our general heuristic proof procedures.

Beside Horn abstractions, tableau based methods on suitable first-order abstractions should also yield better estimate for the overall search process than theorem cardinality. Obviously, there is much space for the design of elaborated on-line and off-line heuristics. We expect a large impact in form of a transfer from AP, e.g. by suiting relaxed plans and pattern databases to ATP. Because the apparent differences in the input, the transfer will certainly not be one-to-one. On the other hand, the process of *proof-finding* and *proof-refinement* will possibly carry over from different design phases in MC.

# Part V

# Software Verification

# Paper 13

# Inferring Flow of Control in Program Synthesis by Example

Stefan Schrödl and Stefan Edelkamp.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science. Springer, pp. 171–182, 1999.

**Abstract**

We present a supervised, interactive learning technique that infers control structures of computer programs from user-demonstrated traces. A two-stage process is applied: first, a minimal deterministic finite automaton (DFA) $M$ labeled by the instructions of the program is learned from a set of example traces and membership queries to the user. It accepts all prefixes of traces of the target program. The number of queries is bounded by $O(k \cdot |M|)$, with $k$ being the total number of instructions in the initial example traces. In the second step we parse this automaton into a high-level programming language in $O(|M|^2)$ steps, replacing jumps by conditional control structures.

# 13.1 Introduction

## 13.1.1 Program Synthesis from Examples

The ultimate goal of program synthesis from examples is to teach the computer to infer general programs by specifying a set of desired input/output data pairs. Unfortunately, the class of total recursive functions is not identifiable in the limit [148]. For tractable and efficient learning algorithms either the class has to be restricted or more information has to be provided by a cooperative teacher.

Two orthogonal strains of research can be identified [127]. Until the late 1970s, the focus was on inferring functional (e.g., Lisp) programs based on traces. Since the early 1980s the attention shifted towards model-based and logic approaches.

All functional program synthesis mechanism are based on two phases: *trace generation* from input/output examples, and *trace generalization* into a recursive program. Biermann's *function merging mechanism* [37] takes a one-parameter Lisp function whose only predicate is *atom*, and decomposes the output in an algorithmic way into a set of nested basic functions. Subsequently, they are merged into a minimal set that preserves the original computations by introducing discriminant predicates. These mechanisms perform well on predicates that involve structural manipulation of their parameters, such as list concatenation or reversal. However, their drawbacks are two-fold. The functional mapping between input and output terms cannot be determined in this straightforward way for less restrictive applications; on the other hand, manually feeding the inference algorithm with example traces can be a tedious and error-prone task. Secondly, the merging algorithms require exponential time in general.

The second direction of research (frequently called *Inductive Logic Programming*) is at the intersection between empirical learning and logic programming. A pioneering work was Shapiro's *Model Inference System* [325] as a mechanism for synthesizing Prolog programs from positive and negative facts. The system explores the search space of clauses using a configurable strategy. The subsumption relation assists in specializing incorrect clauses implying wrong examples, and in adding new clauses for uncovered ones. The critical issues are the undecidability of subsumption in the general case, the large number of required examples, and the huge size of the search space.

## 13.1.2 Programming in the Graphical User Interface

The last decades have seen a revolutionary change in human-computer interfaces. Instead of merely typing cryptic commands into a console, the user is given the illusion of moving around objects on a "desktop" he already knows from his everyday-life experience. Users can refer to an action by *simply performing* the action, something they already know how to do. Therefore, they can more easily handle *end user programming* tools.

Many spreadsheet programs and telecommunication programs have built-in *macro recorders*. Similarly to a tape recorder, the user presses the "record" button, performs a series of keystrokes or mouse clicks, presses "stop", and then invokes "play" to replay the entire sequence. Frequently, the macro itself is internally represented in a higher programming language (such as Excel macros in Visual Basic).

Moreover, the current trends in software development tools show that even programming can profit from graphical support. "Visual computing" aims at relieving conven-

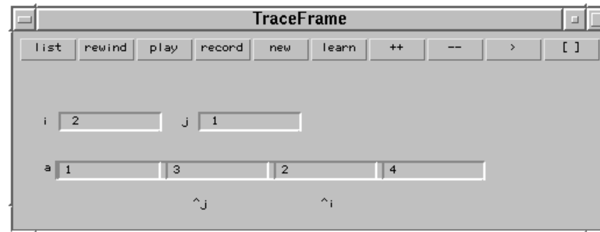


Figure 13.1: Trace Frame.

tional programming from the need of mapping a visual representation of objects being moved about the screen into a completely different textual representation of those actions. In an ideal general-purpose programming scenario, we could think of a domain-independent graphical representation for standard data structures, such as arrays, lists, trees, etc. which can be visually manipulated by the user.

Cypher gives an overview of current approaches [76]. Lieberman's *Tinker* system permits a beginning programmer to write Lisp programs by providing concrete examples of input data, and typing Lisp expressions or providing mouse input that directs the system how to handle each example. The user may present multiple examples that serve to incrementally illustrate different cases in conditional procedures. The system subsequently prompts the user for a distinguishing test. However, no learning of program structures takes place.

Based on these observations, we argue that program synthesis from traces could regain some attraction. The burden of trace generation can be greatly alleviated by a graphical user interface and thus becomes feasible.

In this paper, we propose an efficient interactive learning algorithm which solves the complexity problem of the merging algorithm in functional program synthesis. Contrary to the latter approach, we focus on imperative programming languages. They also reflect more closely the iterative nature of interaction with graphical user interfaces. The flow of control in imperative languages is constituted by conditional branches and loops; their lack in most current macro recorders is an apparent limitation.

## 13.2 Editing a First Example Trace

Figure 13.1 shows our prototypical graphical support. The user generates a first example trace by performing a sequence of mouse selections, mouse drags, menu selections, and key strokes.

Throughout the paper, we will exemplify the inference mechanism with the well-known *bubble-sort* algorithm. The user might start with the sample array $\texttt{a} = [2, 1]$ of length $\texttt{n} = 2$. A variable `i` is introduced to hold the number of remaining iterations, and is initialized to one (`int i=n-1`). Then he states that the end is not yet reached (`i>0`). Subsequently he initializes another variable `j` to zero, meant as an index for traversing the array (`int j=0`). Now the array element with index 0 is compared to its successor (`a[j]>a[j+1]`). Since the comparison 1>2 fails (`F`) he swaps the elements (`swap(j,j+1)`). For ease of exposition, we assume that the `swap`-procedure has already been programmed to interchange two values in the array. The user increases `j`

(`j++`) and then observes that the array has been traversed up to position `i` (`j<i; F`) in which case `i` is decremented (`i-`). The next iteration starts. But since `i` now has reached the left border (`i>0; F`) the sorting is accomplished and the procedure stops (`return`). In summary, the example generated by the end user is given as follows: `i=0; i>0; T; j=0; a[j]<a[j+1]; F; swap(j,j+1); j++; j<i; F; i-; i>0; F; return`.

## 13.3 The ID-Algorithm

*Grammar inference* is defined as the process of learning an unknown grammar given a finite set of labeled examples. An important, widely used subset of formal languages are *regular grammars*, which can be generated and recognized by deterministic finite automata (DFA). However, given a finite set of positive examples and a finite, possibly empty set of negative examples, the problem of learning a minimum state DFA equivalent to the target is *NP*-hard [149]. Hence, the learner's task has to be simplified by imposing certain desired criteria on the examples (like structural completeness, characteristic samples), or by providing the learner with access to sources of additional information, like a knowledgeable teacher (oracle) who responds to queries generated by the learner.

Our algorithm is based on Angluin's ID-algorithm which is briefly recalled in this section. It may be skipped in a first reading.

Let $\Sigma$ be the set of symbols, $\Sigma^*$ be the set of strings, and $\lambda$ be the empty string. Furthermore, let $M = (Q, \delta, \Sigma, q_0, F)$ be a DFA according to the usual quintuple definition and $L(M)$ be the language accepted by $M$. A state $q$ in $M$ is *alive* if it can be reached by some string $\alpha$ and left with some string $\beta$ such that $\alpha\beta \in L(M)$. In a minimal DFA there is only one state $d_0$ that is not alive. A set of strings $P$ is said to be *live-complete* w.r.t. $M$ if for every live state $q$ in $M$ there exists a string $\alpha \in P$ such that $\delta(q_0, \alpha) = q$. Therefore, $P' = P \cup \{d_0\}$ represents all states in $M$. In order to find a string representation of the state reached on reading an input $b$ from the state represented by $\alpha$ we define a function $f : P' \times \Sigma \to \Sigma^* \cup \{d_0\}$ by $f(d_0, b) = d_0$ and $f(\alpha, b) = \alpha b$. The *transition set* $T'$ denotes the set of all elements of $P'$, together with all elements $f(\alpha, b)$ for all $(\alpha, b) \in P \times \Sigma$. Analogously to $P$ we define $T = T' - \{d_0\}$.

The goal of the ID algorithm (Figure 13.2) is to construct a partition of $T'$ that places all the equivalent elements in one state [11]. The equivalence relation is the Nerode relation such that the resulting DFA will be minimal [4]. The algorithm starts with an initial partition of one accepting and one non-accepting state and refines it successively. In each step $i$ of ID a string $v_i$ is drawn such that for any two states $q$ and $q'$ there exists a $j \leq i$ with $\delta(q, v_j) \in F$ and $\delta(q', v_j) \notin F$ or vice versa. Thus, we define the $i$-th partition $E_i$ as follows: $E_i(d_0) = \emptyset$ and $E_i(\alpha) = \{v_j | j \leq i, \alpha v_j \in L(M)\}$. Then for every two strings $\alpha, \beta \in T$ with $\delta(q_0, \alpha) = \delta(q_0, \beta)$ we have $E_j(\alpha) = E_j(\beta)$ for all $j \leq i$. For each $i$ the algorithm searches for a separating pair $\alpha, \beta$ and a symbol $b$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$. Let $\gamma$ be any string that is either in $E_i(f(\alpha, b))$ and not in $E_i(f(\beta, b))$ or vice versa. Then we define $v_{i+1} = b\gamma$ and construct the $(i + 1)$-th partition as follows. For each $\alpha \in T$ we query the string $\alpha v_{i+1}$. If $\alpha v_{i+1} \in L(M)$ we set $E_{i+1} = E_i \cup \{v_{i+1}\}$; otherwise, we let $E_{i+1} = E_i$ unchanged.

We iterate until no separating pair $\alpha$, $\beta$ exists and extract $M$ from the sets $E_i$ and the transition set $T$ as follows. The states of $M$ are the sets $E_i(\alpha)$, for $\alpha \in T$. The initial

**Input**: a live complete set $P$ and a teacher to answer membership queries
**Output**: a description of the canonical DFA $M$ for the target regular grammar

$i = 0; v_i = \lambda; V = \{\lambda\}, T = P \cup \{f(\alpha, b) | (\alpha, b) \in P \times \Sigma\}; T' = T \cup \{d_0\}, E_0(d_0) = \emptyset;$
**for each** $\alpha \in T$
    **if** $(\alpha \in L)$ $E_0(\alpha) = \{\lambda\}$ **else** $E_0(\alpha) = \emptyset$**;**
**while** ($\exists \alpha, \beta \in P'$ and $b \in \Sigma$ such that $E_i(\alpha) == E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$)
    let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
    let $v_{i+1} = b\gamma$
    let $V = V \cup \{v_{i+1}\}$ and $i = i + 1$
    **for each** $\alpha \in T$
        **if** $(\alpha v_i \in L)$ $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$; **else** $E_i(\alpha) = E_{i-1}(\alpha)$;
Extract the automaton $M$ for $L$ from the sets $E_i$ and $T$ (see text)

Figure 13.2: Angluin's ID-algorithm.

state of $M$ is $E_i(\lambda)$. The accepting states of $M$ are the sets $E_i(\alpha)$, where $\alpha \in T$ and $\lambda \in E_i(\alpha)$. If $E_i(\alpha) = \emptyset$ then we add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$; else we set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$ for all $\alpha \in P$ and $b \in \Sigma$.

Angluin proved that ID asks no more than $n \cdot |\Sigma| \cdot |P|$ queries, where $n$ is the number of states in $M$: the algorithm iterates through the *while*-loop at most $n$ times, since each time at least one set $E_i$ (corresponding to a state) is partitioned into two subsets. It asks $|T|$ questions, where $T$ contains no more than $|\Sigma| \cdot |P|$ elements.

## 13.4 Customizing ID for Program Traces

### 13.4.1 Naive Approach

A simple strategy to apply the ID-algorithm to the problem of program inference from traces goes as follows. The alphabet $\Sigma$ consists of all program lines occurring in the examples. More precisely, we partition $\Sigma$ into $\Gamma \cup \Lambda \cup \Delta \cup \{\texttt{return}\}$, where $\Gamma$ is the set of non-branching instructions (e.g. assignments), $\Lambda$ is the set of (boolean) tests (e.g. numerical comparisons), $\Delta = \{\texttt{T}, \texttt{F}\}$ is the set of boolean values, and `return` signals the end of the procedure. The language $L$ to be learned is regular and consists of all prefixes of valid execution traces. Programs are represented as finite state machines, where transitions are labeled with the respective instructions. Let $Pr(\alpha)$ be the set of all prefixes to $\alpha$. The live-complete set $P$ for the ID-algorithm can now be fixed as $P = Pr(S) \cup \{\lambda\}$, with $S$ being the example trace.

For the initial examples in $P$, the user is free to choose any data, such as the array $[2, 1]$ in our case. As a heuristic guideline, the first examples are supposed both not to be overly lengthy (in order to reduce the number of subsequent questions), but at the same time cover all states of the automaton (in order to specify $P$). However, this requirement is not compulsory: in the version *IID* of the algorithm [282], the initial set of examples need not to be *live-complete*; the user is allowed to incrementally refine the automaton structure by presenting additional (positive or negative) examples later on.

Using this scheme, the number of queries (2158) asked for our bubble-sort case is clearly inacceptably high. Fortunately, the majority of them can immediately be answered by the system itself.

### 13.4.2 Pruning

We make the following general assumptions to hold for all execution traces $\alpha$ in $\Sigma^*$.

1. If $\alpha a \in L$ for some $a \in \Sigma$ then also $\alpha \in L$. In words: every prefix of a word in $L$ is itself in $L$.

2. If $\alpha ab \in L$ and $\alpha ac \in L$ where $a \in \Gamma \cup \Delta$ and $b, c \in \Sigma$, then $b = c$. In words: There is only one instruction that follows a non-branching instruction or a boolean.

3. If $\alpha ab \in L$ and $a \in \Lambda$ then $b \in \Delta$. In words: A test is only followed by a boolean denoting its outcome.

4. We have $\alpha ab \notin L$ for $a =$ `return` and all $b \in \Sigma$. In words: No instruction may follow the end statement.

If condition 3. or 4. is violated, the trace is malformed and is hence rejected.

According to condition 1., we can efficiently store both the example traces and the query traces confirmed by the user in a *trie* data structure [210]. The bold path in Figure 13.3 corresponds to the first example trace of Section 13.2. Given a query string $\alpha b\gamma$, we tentatively insert it into the trie. If it is already contained, the answer is "yes". If the new trie forks at a non-branching instruction, condition 2. is violated and thus the answer is "no". Otherwise, the user is prompted. Unless his response is positive, the query string is removed.

For example at branch (3) in Figure 13.3 the system asks: `int i=n-1; i>0;T; int j=0;`
`a[j]>a[j+1]; T; swap(j,j+1); j++; j<i; T; int i=n-1;` $\in L$? The user will answer "no".

In the further course of the session, the system will eventually "guess" all possible instructions as $b\gamma$ until the correct one `a[j]>a[j+1]` is found. As a further simplification, we can allow the user to edit the question and to immediately type in the right continuation.

### 13.4.3 Selection of Example Data

Ideally, the system should present its queries by animating a sequence of instructions for a suitable instantiation of the variables. Given only the raw code fragments, it might be difficult for the user to find the correct continuation.

This raises the question of how to select data which is consistent with a given trace, i.e., how to find an assignment to the variables that makes one choice point true and another one which makes it false. Two options are conceivable: the user could be asked to give a pool of examples independently of (prior or alternating to) the learning process, from which the system can choose some appropriate one. Alternatively, he can provide a specification to generate random data. E.g., the bubble-sort algorithm should sort every permutation of the array elements, which we w.l.o.g. fix to be $[1, 2, \ldots, n]$. For instance,

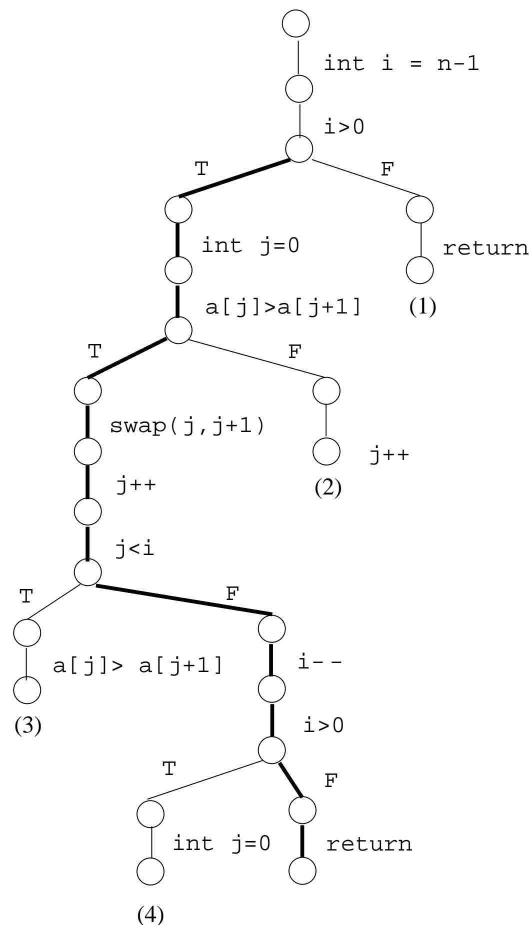


Figure 13.3: Trie of example traces (bold edges) and query results (thin).

the array $a = [3,1,2]$ of length $n = 3$ leads to the following instantiation for question (3): `int i=2; 2=i>0; T; int j=0; 2=a[0]>a[1]=1; T; swap(0,1); j++; 1=j<i=2; T; i=2;` $\in L$? The user responds by replacing `i=2` by the next step which compares $3 = a[1] > a[2] = 2$, i.e., the test `a[j]>a[j+1]`.

Figure 13.4 depicts the finite state machine for the bubble-sort program inferred by the ID-algorithm. All states are accepting, and all omitted transitions lead to the dead state $d_0$.

### 13.4.4 Query Complexity

Every affirmatively answered membership question and every edited answer string inserts at least one node into the trie. Incrementally extending the trie in this way contributes to reduce the number of user questions. The total number is bounded by the size of the final trie minus that of the the initial one. In our bubble-sort example, this bound corresponds to the number of thin edges in Figure 13.3. Actually, the user is asked four instead of 2158 times.

Due to the restrictions on well-formed traces, we can specify a tighter upper bound on the number of user questions, compared to that of Angluin. If $\alpha \in T$ violates one of conditions 2. - 4., then $\alpha v_i \notin L$ for all distinguishing strings $v_i$. We count the number of the remaining valid elements $\tilde{T}$.

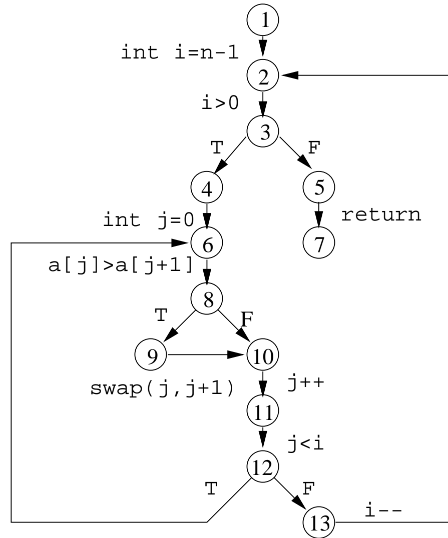


Figure 13.4: DFA for bubble-sort.

We assume that all given examples in $P$ are complete traces, i.e. end with a `return` statement. Therefore, extensions $f(\alpha, b) \in T$ to $\alpha \in P$ are only available at proper prefixes of elements in the example set. However, if $\alpha$ ends with a non-branching instruction, restriction 2. constrains $f(\alpha, b)$ to be in the set $P$. In case $\alpha$ ends with a test instruction, condition 3. leaves us with two choices `T` and `F` for $b$. With $k$ denoting the total number of tests in the example set, we have that $|\tilde{T}|$ is bounded by $k + |P|$. Finally, we conclude that the total number of membership queries is bounded by $n \cdot (k + |P|) = O(n \cdot |P|)$.

## 13.5 Transforming Automata into Structured Programs

It is straightforward to write down any generated automaton as a program using some form of jumps (e.g., *goto*-statements).

For more complex algorithms such flow charts quickly become confusing. In most current high-level programming languages, jump statements are either strongly discouraged (e.g., in C), or do not exist at all (e.g., in Pascal). Instead, high-level constructs are available for conditional branching and looping.

Therefore, we do not regard the automaton generated by the ID-algorithm as the final output, but rather apply a transformation in order to replace jumps by control structures.

Our algorithm transforms the automaton graph step by step by repeatedly collapsing a subgraph into a new edge, for which we keep track of extra information: its type (e.g., simple, test, sequence, *while*-loop, etc.), possibly its subcomponents, and the set of its successors.

Two adjacent edges labeled with arbitrary instructions other than tests or booleans can be merged into a *sequence* if they are the only inward or outward edges of the enclosed node. Connected tests can be merged into (compound) *conditions* containing boolean operators depending on the role of their `T`- and `F`-edges in the obvious way. For example, if test $t_1$ is connected to test $t_2$ via its `T`-edge, and the `F`-edges of both $t_1$ and $t_2$ point to the same node $v$, then a compound condition $t_1 \wedge t_2$ is formed whose `T`-edge leads to the

```
        int i=10, j=20;
l1 :    i--;
l2 :    if (j==0) return;
        j--;
        if (i > 0 && j>0) goto l1;
        goto l2;
```

Figure 13.5: Without semantic information unfolding is impossible.

same node pointed to by the T-edge of $t_2$, and whose F-successor-node is $v$.

The more interesting cases are the instances where control structures are inferred: a (simple or compound) condition whose T-edge leads to a non-test edge with successor node $v$, and whose F-successor-node is also $v$, can be merged into an *if-then*-statement pointing to $v$. Similarly, if the T- and the F-edge lead to different edges with the same successor node, then the resulting conditional statement additionally contains an else-part. A *while*-loop is a condition-edge $c$ whose T-successor leads to an edge (i.e., the repeated block) which has, in turn, $c$ as its successor. The resulting edge points to the destination of the F-successor-edge of $c$. If the two edges are interchanged, the condition in the generated *while*-statement is negated. In *do-while*-loops, the condition follows the edge for the repeated block.

First, the algorithm initializes the in-degrees of all nodes (in linear time). Then all $n$ nodes are repeatedly checked for applicable transformation rules. If none is found, we are done; otherwise the automaton is altered accordingly, and the degree of affected nodes is adjusted. Both these operations require constant time. Since each transformations removes at least one node, at most $n$ iterations are performed, giving an overall worst-case complexity of $O(n^2)$.

Note that, in principle, it is not always possible to transform jumps into control structures without reasoning about the semantics of a program or changing the set of variables (Fig. 13.5 sketches a critical loop structure). In these cases, the system should at least try to minimize the number of remaining *goto*s. Such graceful degradation is not covered by our algorithm and left as a topic for further research.

For our example, Figure 13.6 show the sequence of transformations applied to the original automaton of Figure 13.4. First, the edges $(6, 8)$, $(8, 9)$, $(8, 10)$, and $(9, 10)$ are collapsed into an *if-then*-statement (a). In the next step, the edge labeled `j++` is appended to form a *sequence* edge (b). Now we create a *do-while*-loop, since the test edge $(11, 12)$ appears after the repeated block (c). The two next steps summarizes it, together with the edges with respective labels `int j=0` and `i-`, into a sequence (d). We create the outer *while*-loop (e), and then concatenate `int i=n-1` and `return` to it, such that only one edge is left corresponding to the final program (f).

## 13.6 Conclusion and Discussion

We have presented a supervised, interactive learning algorithm which infers control structures of computer programs from example traces and queries.

First, a deterministic finite automaton is learned by a customized version of the ID-

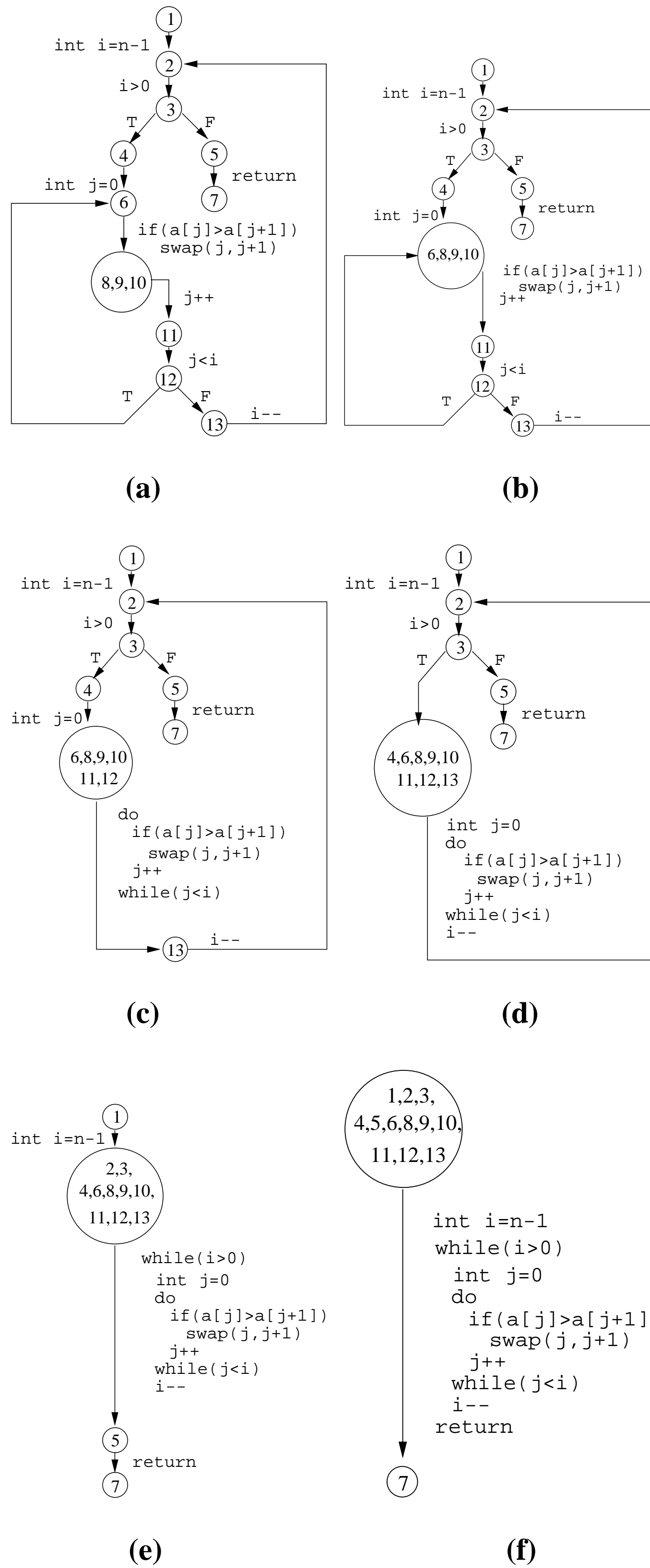


Figure 13.6: Transformation of the DFA into a structured program.

algorithm for regular language inference. By exploiting the syntactical form of programs and allowing the user to incrementally type in instructions, the number of questions is reduced from an infeasible to a moderate scale. An upper bound of $O(n \cdot |P|)$ membership queries is given. Secondly, the resulting automaton is rewritten in a high-level language with control structures using an $O(n^2)$ algorithm.

An early precursor of this work similar in spirit is presented by Gaines [136]. His approach infers a DFA by exhaustive and exponential search until an automaton is found that is consistent with the given traces.

Schlimmer and Hermens describe a note-taking system that reduces the user's typing effort by interactivly predicting continuations in a button-box interface [318]. An unsupervised, incremental machine learning component identifies the syntax of the input information. To avoid intractability the class of target languages is constrained to so-called $k$-reversible regular languages for which Angluin proposed an $O(n^3)$ inference algorithm [12]. However, for general proposed languages this class is too restrictive. It is not hard to find simple programs not covered by zero-reversible FSM's (as in the examples given in the paper). On the other hand, simply fixing $k$ at a larger value sacrifies minimality of the generated automaton. Schlimmer and Hermens improve the system's accuracy by adding a decision tree to each state. However, prediction is not relevant to our approach since traces are deterministic: A new training example leads to a new FSM.

End users without programming knowledge can take benefit from inference of control structures. More powerful customization tools (e.g., macro recorders) are able to support them in solving more of the repetitive routine work which often needs elementary conditional branching and looping.

For the experienced programmer, the proposed inference mechanism might support the process of software development, mainly in view of integrity and incremental extensibility.

The final set of execution traces (as depicted by the resulting trie) uniquely determines the structure of the automaton. All source fragments in the generated program have been exercised in at least one example. Therefore, no untested code can arise. For recorded execution traces on concrete sample data, differences between the intended and the actual meaning of the program will occur by far more infrequently than bugs in programs developed without the control of explicit variable instantiations. In a way, both stages in the software development cycle, coding and testing, are performed more efficiently in parallel rather than in the usual alternating way.

A sequence of examples should start with simple examples and build to more complex and exceptional cases. Recursive and conditional procedures can be developed incrementally by starting with simple, “incorrect” definitions, and later adding more instances to handle more complicated and special purpose situations. Maintaining all used examples and only adding to this set ensures that previous examples are still covered and that with growing complexity, no new bugs are introduced for cases which have already been successfully treated.

# Paper 14

# Directed Explicit-State Model Checking in the Validation of Communication Protocols

Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: {edelkamp,lafuente,leue}@informatik.uni-freiburg.de

In *International Journal on Software Tools and Technology Transfer*, 2002.

**Abstract**

The success of model checking is largely based on its ability to efficiently locate errors in software designs. If an error is found, a model checker produces a trail that shows how the error state can be reached, which greatly facilitates debugging. However, while current model checkers find error states efficiently, the counterexamples are often unnecessarily lengthy, which hampers error explanation. This is due to the use of "naive" search algorithms in the state space exploration.

In this paper we present approaches to the use of heuristic search algorithms in explicit-state model checking. We present the class of A* directed search algorithms and propose heuristics together with bitstate compression techniques for the search of safety property violations. We achieve great reductions in the length of the error trails, and in some instances render problems analyzable by exploring a much smaller number of states than standard depth-first search. We then suggest an improvement of the nested depth-first search algorithm and show how it can be used together with A* to improve the search for liveness property violations. Our approach to directed explicit-state model checking has been implemented in a tool set called HSF-SPIN. We provide experimental results from the protocol validation domain using HSF-SPIN.

## 14.1 Introduction

Model Checking [65] is a formal analysis technique that has been developed to automatically validate[1] functional properties for software or hardware systems. The properties are commonly specified using some sort of a temporal logic or using automata. There are two primary approaches to model checking. First, *symbolic* model checking [259] uses a symbolic representation for the state set, usually based on binary decision diagrams. Property validation in symbolic model checking amounts to symbolic fixpoint computation. *Explicit state* model checking uses an explicit representation of the system's global state graph, usually given by a state transition function. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure, and property validation amounts to a partial or complete exploration of the state space. In this paper we focus on explicit state model checking and its application to the validation of communication protocols. The protocol model we consider is that of collections of extended communicating finite state machines as described, for instance, in [43] and [151]. Communication between two processes is either realized via synchronous or asynchronous message passing on communication channels (queues) or via global variables. Sending or receiving a message is an event that causes a state transition. The system's global state space is generated by the asynchronous cross product of the individual communicating finite state machines (CFSMs). We follow the Promela computational model [187].

The use of model checking in system design has one great advantage over the use of deductive formal verification techniques. Once the requirements are specified and the model has been programmed, model checking validation can be implemented as a push-button process that either yields a positive result, or returns an error trail. Two primary strategies for the use of model checking in the system design process can be observed.

- *Complete validation* is used to certify the quality of the product or design model by establishing its absolute correctness. However, due to the large size of the search space for realistic systems it is hardly ever possible to explore the full state space in order to decide about the correctness of the system. In these cases, it either takes too long to explore all states in order to give an answer within a useful time span, or the size of the state space is too large to be stored within the bounds of available main memory.

- The second strategy, which also appears to be the one more commonly used, is to employ the model checker as a *debugging aid* to find residual design and code faults. In this setting, one uses the model checker as a search tool for finding violations of desired properties. Since complete validation is not intended, it suffices to use hashing-based partial exploration methods that allow for covering a much larger portion of the system's state space than if complete exploration is needed.

When pursuing debugging, there are some more objectives that need to be addressed. First, it is desirable to make sure that the length of the counterexample is short, so that error trails are easy to interpret. Second, it is desirable to guide the search process to

[1] Within the scope of this paper we use the word “validation” to denote the experimental approach to establishing the correctness of a piece of software, while we use the word “verification” to denote the use of formal theorem proving techniques for the same purpose.

quickly find a property violation so that the number of explored states is small, which means that larger systems can be debugged this way. To support these objectives we present our approach to *directed model checking*, i.e. model checking combined with heuristic search.

Our model-checker HSF-SPIN extends the SPIN framework with various heuristic search algorithms to support directed model checking, e.g. A* [161] and iterative deepening A* [213]. Experimental results show that in many cases the number of expanded nodes and the length of the counter-examples are significantly reduced. HSF-SPIN has been applied to the detection of deadlocks, invariant and assertion violations, and to the validation of LTL properties. In most instances the estimates used in the search are derived from the properties to be validated, but HSF-SPIN also allows some designer intervention so that targets for the state space search can be specified explicitly in the Promela code.

We propose an improvement of the nested depth-first search algorithm that exploits the structure of never claims. For a broad subset of the specification patterns described in [87], such as *Response* and *Absence*, the proposed algorithm performs less transitions during state space search and finds shorter counterexamples compared to classical nested depth-first search. Given a Promela *never claim* $A$ the algorithm automatically computes a partitioning of $A$ in linear time with respect to the number of states in $A$. The obtained partitioning into non-, fully and partially accepting strongly connected components will be exploited during state space exploration.

**Precursory Work.** Much of the content of this paper is a revision of work that was first published in [109] and [107]. The former paper considers safety property analysis for simple protocols. The latter paper extends this work by providing an approach to validating LTL-specified liveness properties and experimenting with a larger set of protocols. Previously unpublished results include the correctness result for the improved nested depth-first search algorithm as well as an extended experimental evaluation of our approach.

**Structure of Paper.** In Section 14.2 we review automata-based model checking. Section 14.3 introduces into directed search algorithms, including A*. Heuristic estimate functions to be used in safety property analysis of communication protocols are suggested in Section 14.4. We describe the HSF-SPIN tool set in Section 14.5 and present experimental results for safety properties in Section 14.6. In Section 14.7 we propose an improvement to the nested depth-first search algorithm used in the analysis of liveness properties and show how this algorithm can be combined with heuristic search. Experimental results on liveness property validation are given in Section 14.8. We discuss related work in Section 14.9 and conclude in Section 14.10.

## 14.2 Automata-based Model Checking

In this Section we review the automata theoretic framework for explicit state model checking (c.f. [65]), describe the validation algorithms in use, and present a practical model checker, the SPIN tool set.

Figure 14.1: Büchi automaton for response property (top left) and for its negation (bottom right).

### 14.2.1 Automata-theoretic Framework

Since we model reactive systems with infinite behaviors, the appropriate formalization for words over state sequences of these systems are Büchi automata. They inherit the syntactic structure of finite state automata but have a different acceptance condition. An infinite run of a Büchi automaton $\mathcal{A}$ over an alphabet $\Sigma_{\mathcal{A}}$ of state symbols is accepting if the set of elements of $\Sigma_{\mathcal{A}}$ that appear infinitely often in the run has a non-empty intersection with the set of accepting states of $\mathcal{A}$. This extends to finite runs by assuming that the final state will be repeated forever. The language $L(\mathcal{A}) \subseteq \Sigma_{\mathcal{A}}^*$ consists of all accepting runs of $\mathcal{A}$. It is sometimes helpful to specify requirements on reactive systems by using some form of a Temporal Logic. In this paper we use Linear Time Temporal Logic (LTL) as defined in [252]. In LTL, the operator $\Box$ represents the modality *globally* ($G$) and the operator $\Diamond$ represents the modality *eventually* ($F$).

In automata-based Model Checking we are interested in determining whether the system $M$, represented by Büchi automaton $\mathcal{B}$, satisfies a property specification $S$, given by another Büchi automaton $\mathcal{A}$. $\mathcal{A}$ can either be given directly, or it can be automatically derived from an LTL property specification. While this derivation is exponential in the size of the formula, typical property specifications result in small LTL formulae so that this complexity is not a practical problem. The Büchi automaton $\mathcal{B}$ satisfies $\mathcal{A}$ iff $L(\mathcal{B}) \subseteq L(\mathcal{A})$. This is equivalent to $L(\mathcal{B}) \cap \overline{L(\mathcal{A})} = \emptyset$, where $\overline{L(\mathcal{A})}$ denotes the complement of $L(\mathcal{A})$. Note that Büchi automata are closed under complementation. In practice, $\overline{L(\mathcal{A})}$ can be computed more efficiently by deriving a Büchi automaton from the negation of an LTL formula. Therefore, in the SPIN validation tool LTL formulae representing a desired property are first negated, and then translated into an equivalent Büchi automaton. In the terminology of the SPIN model checker [189] and its Promela input language this automaton is called a *never claim*, and we will adopt this terminology throughout this paper.

As an example we consider the commonly used *response* property which states that, whenever a certain request event occurred, a response event will eventually follow. Assume that the state following the occurrence of the request is represented by the state predicate $p$, and that a state following the response is denoted by $q$. The corresponding LTL formula is $\phi : \Box(p \to \Diamond q)$ and its negation is $\neg\phi : \Diamond(p \wedge \Box\neg\, q)$. The Büchi automaton and the corresponding Promela never claim for the negated response property are illustrated in Figure 14.1.

The emptiness of $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ is determined using an on-the-fly algorithm based on the synchronous product of $\mathcal{N}$ and $\mathcal{B}$, where $L(\mathcal{N}) = \overline{L(\mathcal{A})}$. Assume that $\mathcal{N}$ is in state $s$ and $\mathcal{B}$ is in state $t$. $\mathcal{B}$ can perform a transition out of $t$ if $\mathcal{N}$ has a successor state $s'$ of $s$ such that the label of the edge from $s$ to $s'$ represents a proposition satisfied in $t$. A run of

**Nested-DFS**$(s)$
  $hash(s)$
  **for all** successors $s'$ of $s$ **do**
    **if** $s'$ not in the hash table **then Nested-DFS**$(s')$
  **if** $accept(s)$ **then Detect-Cycle**$(s)$

**Detect-Cycle**$(s)$
  $flag(s)$
  **for all** successors $s'$ of $s$ **do**
    **if** $s'$ on *Nested-DFS*-Stack **then**
      **exit** *LTL-Property violated*
    **else if** $s'$ not flagged **then Detect-Cycle**$(s')$

Figure 14.2: Nested Depth-First Search

the synchronous product is accepting if it contains a cycle through at least one accepting state of $\mathcal{N}$. $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ is empty if the synchronous product does not have an accepting run.

We use the standard distinction of safety and liveness properties. Safety properties refer to states, whereas liveness properties refer to paths in the state transition diagram. Safety properties can be validated through a simple depth-first search on the system's state space, while liveness properties require a two-fold nested depth-first search. When property violations are detected, the model checker will return a witness (counterexample) which consists of a trace of events or states encountered.

### 14.2.2 Search Algorithms

For the validation of safety properties a simple complete state graph traversal algorithm is sufficient. This is usually either a depth-first (DFS) or a breadth-first (BFS) search algorithm. When a property violating state is encountered, the search stack contains the witness that will be made available to the user. BFS finds errors with minimal witness length, but is rather memory inefficient. DFS is more memory efficient, but tends to produce witnesses of non-optimal length.

Since liveness properties refer to execution paths, a different search approach is needed. The detection of liveness property violations entails searching for accepting cycles in the state graph. This is typically achieved by nested depth-first search (Nested-DFS) that can be implemented with two stacks as shown in Figure 14.2. As for safety properties, the search stacks will be used to construct the witness. In case a property violation is discovered, the first stack will contain the path into an accepting state, while the second stack will illustrate the cycle through the accepting state.

### 14.2.3 The Model Checker SPIN

SPIN [189] is a model checking tool implementing the above discussed approach to automata-based model checking. Its input language Promela permits the definition of

concurrent processes, called *proctypes* in Promela parlance, as well as synchronous or asynchronous communication channels and a limited set of C-like data structures. Concurrency in SPIN is interpreted using an interleaving approach. Properties can be specified in various ways. To express safety properties, the Promela code can be augmented with assertions or deadlock state characterizations. In order to express liveness properties, Promela models can be extended by never claims that express undesired properties of the model. SPIN also provides an automatic linear temporal logic (LTL) to never claim translator. SPIN implements the synchronous product construction approach to determine the emptiness of the intersection of the Promela model and the never claim. SPIN uses on-the-fly state space exploration algorithms, and implements various optimizations such as, for instance, partial order reduction. Promela models can be simulated randomly, user-guided or following an error trail. SPIN has a line-oriented as well as a graphical user interface, called XSPIN. For a more detailed discussion of SPIN we refer to the literature on the SPIN web site[2].

### 14.2.4 Error Trails

If property violations are found, error trails contain important debugging information. Succinctness of these trails is essential for an easy comprehension of the discovered design faults. Lengthy trails can impede proper error trail interpretation.

We illustrate the impact of long error trails with the following example. We refer to the preliminary design of a Plain Old Telephony System (POTS) that we first presented in [202]. This model was generated with the visual modeling tool VIP. It is a "first cut" implementation of a simple two-party call processing, and we know that it is full of faults of various kind. However, in [202] we used SPIN to show that this model is actually capable of connecting two telephones. The model consists of two user processes `UserA` and `UserB` representing the environment behaviour of the switch, as well as two phone handler processes `PhoneHA` and `PhoneHB` representing the software instances that control the internal operation of the switch according to signals (on-hook, off-hook, etc.) received from the environment. Due to space constraints we have to rely on an intuitive understanding of call processing behaviour and the type of signals that are used, for a more detailed description we refer to [202].

Our objective now is to use SPIN in order to debug the POTS model. We are first interested in knowing whether certain inconsistent global system states are reachable. For instance, such an inconsistent state is reached when all user processes and one phone handler process are in *conversation* states, indicating that they presume the two phones to be connected, while the second phone handler is not in a conversation state. Let $p$ and $q$ denote state propositions that are true when phone handlers A and B are in the conversation state, respectively. Let $r$ and $s$ denote state propositions representing the fact that phones A and B are in the conversation state, respectively. The absence requirement for this inconsistent global system state, which is a safety property, can be characterized by the LTL formula

$$\neg\Diamond(p \wedge \neg q \wedge r \wedge s).$$

We used SPIN to validate this property. It turns out not to be valid and SPIN produces an error trail leading into a global system state violating the property as partially illustrated

[2] `netlib.bell-labs.com/netlib/spin`.

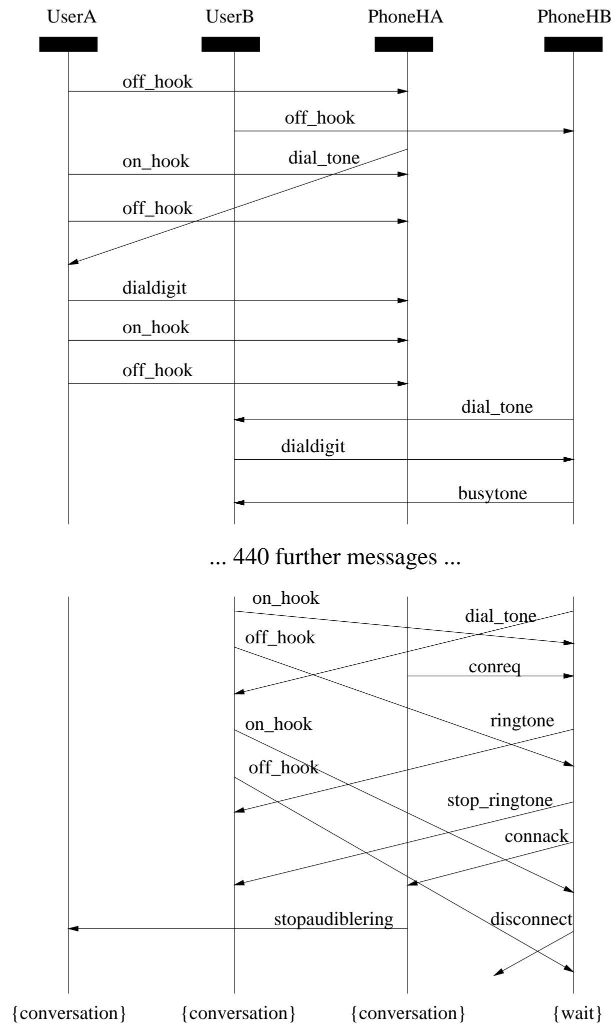


Figure 14.3: POTS example, error trail produced by SPIN. Names in curly brackets denote local control states reached at the end of the trail.

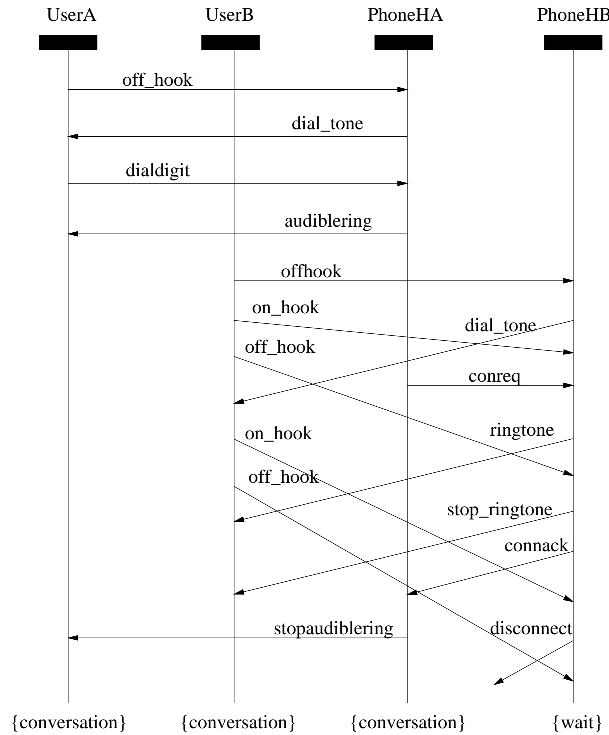


Figure 14.4: POTS example, manually generated shorter error trail.

in Figure 14.3. For the engineer experienced in analyzing call processing sequences it becomes clear that the undesired state is reachable because of race conditions and a lack of synchronization between the `UserB` and the `PhonHB` processes, which probably calls for using synchronous communication at this interface. On the other hand, the error trail that SPIN produces has a length of 2,765 steps and comprises 462 message exchanges - it is obvious that analyzing a trail of that length to locate the cause of an error is an arduous task. The length of the trail is surprising since using some backward analysis, and when knowing the underlying state machine model, it is easy to come up with a much shorter trail by hand, for instance the trail comprising just 16 messages given in Figure 14.4.

The trail length phenomenon is partly due to the high degree of nondeterminism inside the system which can be attributed to the highly concurrent nature of a telephony switch. Another contributing factor is the search strategy that SPIN uses when exploring the system's state space. Resolution of nondeterminism in Promela is random, but SPIN implements this using a fixed priority scheme based on the lexical structure of the Promela model[3]. SPIN will first explore many execution sequences that do not lead to the establishment of a phone call. This means for instance that one phone calls the other, but then decides to hang up, or both phones try to call each other concurrently, before the call sequence converges towards the successful establishment of a call. The depth-first search strategy that SPIN employs will first try to explore all action variants of the first process, and then try out the next process, and so on. However, the target state would be reached much more quickly if all processes did a few steps so that a phone call was established. In conclusion, SPIN is following a rather uniformed search strategy that neither takes knowledge about the model nor knowledge about the property to be validated into

[3]Roughly speaking, this means the lexically first transition in the “first” proctype instance is preferred over other concurrently enabled transitions.

account when deciding which of the possible successor states to explore first. If, however, the state space of the PhoneA, PhoneB and PhoneHA processes were explored in such a way that every state transition brought them nearer to their own local conversation state and if PhoneHB avoided the conversation state, and if globally such transitions were preferred over non-approximating transitions, then a much shorter error trail into the property violating state could be expected. It is the objective of this paper to present guided search algorithms using heuristic guidelines in the state exploration similar to the one just described. When discussing experimental results, we will see that for the POTS example the automatically obtained shortest error trail is 1.5 orders of magnitude shorter than the one generated by SPIN's exploration.

## 14.3 Heuristic Search Algorithms

In this Section we introduce heuristic search algorithms as alternatives to complete state space exploration in model checking. We will restrict the discussion to safety property searches and extend the discussion to liveness properties later on in this paper.

### 14.3.1 Depth-First, Breadth-First and Best-First Search

The detection of a safety property violation is equivalent to finding a state in which the property is violated. The algorithms used for finding the property violating states are typically depth-first and breadth-first searches. Depth-first search (DFS) is memory efficient, but does not provide optimal solutions. Breadth-first search (BFS), on the other hand, is complete and optimal but very inefficient.

State space exploration in model checking safety properties can be understood as a search for a path to a failure state in the underlying problem graph. Since this graph is implicitly generated by node expansions, in contrast to ordinary graph algorithms the search terminates once a target state has been found. BFS and DFS explore the state space without additional knowledge about the search goal. The selection of a successor node in these algorithms is following a fixed, deterministic selection scheme. Heuristic search algorithms, however, take additional search information in form of an estimation function into account. This function returns a number representing the desirability of expanding a node. When the nodes are ordered so that the one with the best evaluation is expanded first and if the evaluation function estimates the cost of the cheapest path from the current state to a desired one, the resulting greedy best-first search (BF) often finds solutions fast. However, it may suffer from the same defects as depth-first search – it is not optimal and the search may be stuck in dead ends or local minima.

### 14.3.2 Algorithm A*

Algorithm A* [161] combines best-first and breadth-first search for a new evaluation function $f(u)$ by summing the generating path length $g(u)$ and the estimated cost $h(u)$ of the cheapest solution starting from $u$. Figure 14.5 displays the effect of A* compared to DFS, BFS and BF and Table 14.1 depicts the algorithm in pseudo code. The node expansion of $u$ is indicated by access to the successor set $\Gamma(u)$. The set *Closed* denotes the set of all already expanded nodes and the list *Open* contains all generated but not yet expanded

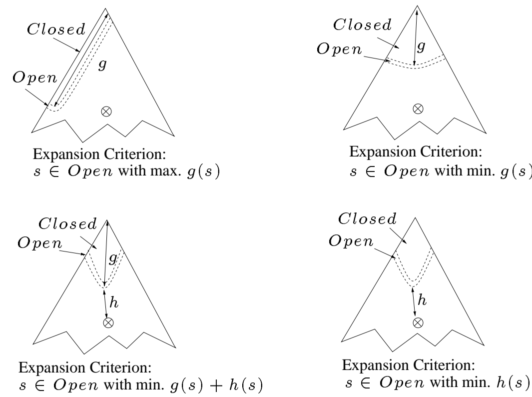


Figure 14.5: Different search strategies: DFS (top left), BFS (top right), A* (bottom left) and BF (bottom right)

nodes. Similar to Dijkstra's single-source shortest path algorithm [80], A* successively extracts the node $u$ with minimal merit $f(u)$ from the set *Open* and terminates if this node represents a failure state.

As the combined merit $f(u) = g(u) + h(u)$ merely changes the ordering of the nodes to be expanded, on finite problem graphs A* is complete. Moreover, by changing the weights of the edges in the problem graph from 1 to $1+h(v)-h(u)$, it can also be observed that A* in fact performs the same computation as Dijkstra's single-source shortest-path algorithm on the re-weighted graph. If for all edges $(u, v)$ we have $1 + h(v) - h(u) \geq 0$, optimality of A* is inherited from the optimality of Dijkstra's algorithm. It can also be shown that the path length for every expanded node is optimal, so that we correctly terminate the search at the first target node.

If $1 + h(v) - h(u) < 0$, negatively weighted edges affect the correctness proof of Dijkstra's algorithm. In this case we have $f(u) + 1 + h(v) - h(u) < f(v)$ such that nodes that have already been expanded might be encountered on a shorter path. Contrary to Dijkstra's algorithm, A* deals with them by possibly re-inserting nodes from the set of already expanded nodes into the set of *Open* nodes (re-opening). On every path from $s$ to $u$ the accumulated weights in the two graph structures differ by $h(s)$ and $h(u)$ only. Consequently, re-weighting cannot introduce negatively weighted cycles so that the problem remains (optimally) solvable. One can show that given a lower bound estimate (admissible heuristic) the solution returned by the A* algorithm with re-opening is indeed a shortest one [115]. The main argument is that there is always a correctly estimated node on an optimal path in the set *Open*. This node has to be considered before expanding any non-optimal goal node.

Figure 14.6 depicts the impact of heuristic search in a grid graph. If $h$ is the trivial constant zero function, A* reduces to Dijkstra's algorithm, which in case of uniform graphs further collapses to BFS. Therefore, starting with $s$ all depicted nodes shown are generated until the goal node $t$ is expanded. If we use $h(u)$ as the Euclidean distance to node $t$, then only the nodes in the hatched region are ever removed from the *Open* set.

```
A*(s)
  Open ← {}; Closed ← {}; f(s) ← h(s);
  Insert(Open, s, f(s))
  while (Open ≠ ∅)
    u ← Deletemin(Open); Insert(Closed,u)
    if (failure(u)) exit Safety Property Violated
    for all v in Γ(u)
      f'(v) ← f(u) + 1 + h(v) − h(u)
      if (Search(Open, v))
        if (f'(v) < f(v))
          DecreaseKey(Open, v, f'(v))
      else if (Search(Closed, v))
        if (f'(v) < f(v))
          Delete(Closed, v); Insert(Open, v, f'(v))
      else Insert(Open, v, f'(v))
```

Table 14.1: The A* algorithm searching for violations of safety properties.

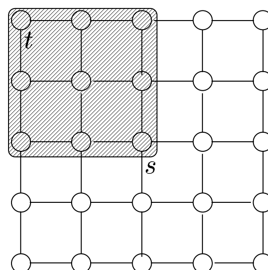


Figure 14.6: The effect of heuristic search in a grid graph.

```
IDA*(s)
  Push(S, s, h(s)); U ← U′ ← h(s)
  while (U′ ≠ ∞)
    U ← U′; U′ ← ∞
    while (S ≠ ∅)
      (u, f(u)) ← Pop(S)
      if (failure(u)) exit Safety Property Violated
      for all v in Γ(u)
          if (f(u) + 1 − h(u) + h(v) > U)
            if (f(u) + 1 − h(u) + h(v) < U′)
              U′ ← f(u) + 1 − h(u) + h(v)
          else
            Push(S, v, f(u) + 1 − h(u) + h(v))
```

Table 14.2: The IDA* algorithm searching for violations of safety properties.

### 14.3.3 Iterative Deepening A*

Algorithm A* has one severe drawback. Once the space resources for storing all expanded and generated nodes are exhausted, no further progress can be made. Therefore, the iterative deepening variant of A*, IDA* [213] for short, counterbalances time for space. It traverses the tree expansion of the problem graph instead of the problem graph itself with a memory requirement that grows linear with the depth of the search tree. As shown in the pseudo-code of Table 14.2, IDA* performs a sequence of bounded DFS iterations. In each iteration, it expands all nodes having a total cost not exceeding threshold $U$, which is determined as the lowest cost $U'$ of all generated but not expanded nodes in the previous iteration. IDA* is complete and optimal, since it expands all nodes with an increasing threshold value for each possible merit value. Since the average number of successors is often large, the tree expansion grows exponentially with increasing depth. Therefore, the last iteration in IDA* often dominates the search effort.

Due to the depth-first structure of IDA*, duplicate state expansions may not be detected, resulting in redundancy. Therefore, similar to depth-first and best-first search as long as memory is available, all generated nodes are kept in a transposition table. To allow dynamic updates of node information, for each node in the table the shortest generating path length and the corresponding predecessor are also maintained.

To improve duplicate detection, IDA* can be combined with bit-state hashing [187] which hashes an entire state vector into a single bit wide table. The bit position indicates whether the state has been reached before, or not. In single bit-state hashing, a hash function $h_1$ maps a state $S$ to position $h_1(S)$; $S$ is stored by setting the bit $h_1(S)$ and searched by querying $h_1(S)$. Double bit-state hashing often improves state space coverage by applying a second hash-function $h_2$. A state $S$ is stored in setting $h_1(S)$ and $h_2(S)$ and detected as a duplicate if both bits are set.

Bit-state hashing as shown in Fig. 14.7 implies that a retrieved node might be an unexpected synonym, since there is no way to distinguish a real duplicate from a false

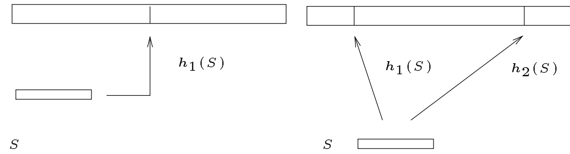


Figure 14.7: Single and double bit-state hashing.

one. False duplicate detection induces an incomplete state space traversal, which can be compensated by different hash functions in different runs. Therefore, re-expanding a duplicate inside IDA* is dangerous, since the information of generating path length and predecessor path length might be false. Subsequently, we avoid reopening and refer to this variant of IDA* as *Partial IDA**. Note that the advantage of Partial IDA* compared to A* is that it can track the solution path on the recursion stack which means that no predecessor link is needed. Reopening in IDA* will not be encountered when the heuristic function is consistent. In this case the priorities $f = g + h$ increase on any generating path, since $f(u) = g(u) + h(u) \leq g(u) + h(u) + 1 + h(v) - h(u) = g(u) + 1 + h(v) = g(v) + h(v) = f(v)$ for all edges $(u, v)$ in the tree expansion of the problem graph. Most practical heuristics satisfy this criterion. The negative impact of partial state space coverage due to bitstate search is reduced by repeating the search with restarts on different hash functions.

## 14.4 Search Heuristics for Safety Properties

In this Section we introduce search heuristics used by our tool HSF-SPIN in the analysis of safety properties for Promela models. We use $S$ to denote a global system state of the model. In $S$ we have a set $\mathcal{P}(S) \subseteq \{P_i \mid i \geq 0\}$ of currently active processes. For the sake of simplicity we assume a fixed number of processes and write $\mathcal{P}$ instead of $\mathcal{P}(S)$. For a process $P_i$ we use $pc_i$ to refer to the current local control state. $T_i$ denotes the set of transitions within the proctype instance $P_i$ and $S_i$ denotes the set of local states of $P_i$.

**Violation of Invariants.** System invariants are state predicates that hold over every global system state $S$. When searching for invariant violations it is helpful to estimate the number of system transitions until a state is reached where the invariant is violated. Given a logical global state predicate $f$, let $H_f(S)$ be an estimation of the number of transitions necessary until a state $S'$ is reached where $f$ holds, starting from state $S$. Similarly, let $\overline{H}_f(S)$ denote the number of transitions necessary until $f$ is violated, which is helpful when validating negations of state predicates. Let $a$ be a Boolean variable, and $g$ and $h$ logical predicates. We give a recursive definition of $H_f$ as a function of $f$, with the first part of the definition given in Figure 14.8.

In the definition of $H_{g \wedge h}$ and $\overline{H}_{g \vee h}$, the use of *plus* $(+)$ suggests that $g$ and $h$ are independent, which may not be true. Consequently, the estimate is not necessarily a lower bound, affecting the optimality condition for A*. Since it is our goal to obtain short but not necessarily optimal paths, we tolerate these inadequacies. To obtain lower bounds, we may replace *plus* $(+)$ with $\max$.

Formulae describing system invariants may contain other terms, such as relational

| $f$ | $H_f(S)$ | $\overline{H}_f(S)$ |
|---|---|---|
| $true$ | $0$ | $\infty$ |
| $false$ | $\infty$ | $0$ |
| $a$ | **if** $a$ **then** $0$ **else** $1$ | **if** $a$ **then** $1$ **else** $0$ |
| $\neg g$ | $\overline{H}_g(S)$ | $H_g(S)$ |
| $g \vee h$ | $\min\{H_g(S), H_h(S)\}$ | $\overline{H}_f(S) + \overline{H}_g(S)$ |
| $g \wedge h$ | $H_g(S) + H_h(S)$ | $\min\{\overline{H}_g(S), \overline{H}_h(S)\}$ |

Figure 14.8: Definition of $H_f$ for Boolean expressions $f$.

| $f$ | $H_f(S)$ |
|---|---|
| $full(q)$ | $capacity(q) - length(q)$ |
| $empty(q)$ | $length(q)$ |
| $q?[t]$ | length of minimal prefix of $q$ without $t$ ($+1$ if $q$ lacks message tagged with $t$) |
| $a \otimes b$ | if $a \otimes b$ then 0, else 1 |
| $f$ | $\overline{H}_f(S)$ |
| $full(q)$ | if $full(q)$ then 1, else 0 |
| $empty(q)$ | if $empty(q)$ then 1, else 0 |
| $q?[t]$ | if $head(q) \neq t$ then 0, else maximal prefix of $t$'s |
| $a \otimes b$ | if $a \otimes b$ then 1, else 0 |

Figure 14.9: Definition of $H_f$ for Boolean queue expressions and relational operators in $f$.

operators and Boolean functions over queues. We extend the definition of $H_f$ and $\overline{H}_f$ as shown in Figure 14.9. The function $q?[t]$ refers to the expression that is true when the message at the head of queue $q$ is tagged with a message of type $t$. All other functions are self-explaining. The symbol $\otimes$ represents relational operators ($=, \neq, \leq, <, >, \geq$).

Note that the estimate is coarse but nevertheless very effective in practice. It is possible to refine these definitions for specific cases. For instance, $H_{a=b}$ can be defined as $a - b$ in case $a \geq b$ and $a$ is only ever decremented and $b$ is only ever incremented. However, we have not pursued these refinements any further.

Another statement that typically appears in system invariants is the *at* predicate which expresses that a process $P$ with a process id $pid$ of a given proctype $PT$ is in its local control state $s$[4]. We will write this as $i@s$, with $s \in S_i$. The corresponding definition is given in Figure 14.10. We use $pc_i$ to express the local state of process $P_i$ in the current global state $S$. The value $D_i(u, v)$ is the minimal number of transitions necessary for the finite state machine $P_i$ to reach state $u$ starting from state $v$, where $u, v \in S_i$. The matrix $D_i$ can be efficiently pre-computed with the all-pairs shortest-path algorithm of Floyd/Warshall in $O(|S_i|^3)$ time [70]. Note that $|S_i|$ is small in comparison to the overall search space.

**Violations of Assertions.** The Promela statement `assert` allows to label the model with logical assertions. Given that an assertion $a$ labels a transition $(u, v)$, with $u, v \in T_i$,

[4]In Promela this is expressed as `PT[pid]@s`.

| $f$ | $H_f(S)$ | $\overline{H}_f(S)$ |
|---|---|---|
| $i@s$ | $D_i(pc_i, s)$ | if $pc_i = s$ 1, else 0 |

Figure 14.10: Definition of $H_f$ for control state predicates in $f$.

| $label(t)$ | $executable(t, S)$ |
|---|---|
| `q?x`, q asynchronous channel | $\neg empty(q)$ |
| `q?t`, q asynchronous channel | $q?[t]$ |
| `q!m`, q asynchronous channel | $\neg full(q)$ |
| `condition c` | $c$ |

Figure 14.11: Function *executable* for asynchronous communication operations and boolean conditions, where `x` is a variable, and `t` is a tag.

then we say $a$ is violated if the formula $f = (i@u) \wedge \neg a$ is satisfied.

**Deadlock Detection.** In concurrent systems, a deadlock occurs if at least a subset of processes and resources is in a cyclic wait situation. In Promela, $S$ is a deadlock state if there is no possible transition from $S$ to a successor state $S'$ and at least one of the processes of the system is not in a *valid endstate*[5]. Hence, no process has a statement that is executable. In Promela, there are statements that are always executable, amongst others assignments, `else` statements, and `run` statements used to start processes. For other statements, such as send or receive operations or statements that involve the evaluation of a guard, executability depends on the current state of the system. For example, a send operation `q!m` is only executable if the queue `q` is not full. The following enumeration describes executability conditions for communication statements over asynchronous channels and for boolean conditions:

1. Asynchronous untagged receive operations (`q?x`, with `x` variable) are not executable if the queue is empty. The corresponding formula is $\neg$ *empty*$(q)$.

2. Asynchronous tagged receive operations (`q?t`, with `t` tag) are not executable if the head of the queue is a message tagged with a different tag than `t` yielding the formula $\neg\ q?[t]$.

3. Asynchronous send operations (`q!m`) are not executable if the queue `q` is full which is indicated by the predicate $\neg full(q)$.

4. Conditions (Boolean expressions) are not executable if the value of the condition corresponding to the term $c$ is false.

The Boolean function $executable$, ranging over tuples of Promela statements and global system states, is summarized for asynchronous operations and boolean conditions in Figure 14.11. Synchronous communication operations (rendezvous send/receive) over a synchronous communication channel are only executable if another process is capable

[5]In Promela, a local control state can be labelled as `end` to indicate that it is a valid end state, i.e., that the system may terminate if the process is in that state.

of executing the inverse communication operation (receive/send) on the same channel. If this is the case both operations are performed as an atomic system transition.

In order to obtain a formula $f$ characterizing the executability of a synchronous send operation `q!x` of a process $P_j$ in a global system state $S$ we proceed as follows. For `q!x` to be executable on a given channel $q$ there must be another process $j$ such that in $S$ process $j$ has an executable inverse `q?x` operation. In other words, the formula describes a disjunction over all processes $i \neq j$ and control locations $u$ of process $i$ such that there is an outgoing transition $(u, v)$ labelled `q?x`:

$$\bigvee_{i=1..n, i\neq j,\ u\in S_i | \exists t=(u,v)\in T_i \wedge label(t)=\mathtt{q?x}} pc_i(S)@u$$

The corresponding formula for a synchronous receive operation is obvious.

We now use $f$ for estimating the number of transitions required to execute a synchronous operation by applying it to the $H_f$ heuristic estimate function that we defined above. As result we will obtain the minimum number of local transitions that every process requires in order to reach a state in which the inverse operation is executable. Obviously, this number is a lower bound for the number of global sytem state transitions necessary to perform the synchronous rendez-vous operation.

The negation of the property $f$ is likely to appear in the characterization of deadlocks. Estimating the number of transitions required for reaching a state where a given synchronous rendez-vous is not enabled will result in computing the sum of $\overline{H}$ for each instance $pc_i(S)@u$. The resulting estimate will be the number of $pc_i(S)@u$ terms that evaluate to true in state $S$. Since for a given $i$ at most one of these terms is true, the estimate will return values between 0 and $i - 1$. In other words the number of transitions required for blocking a given synchronous operation will be estimated as the number of local transitions required for each process to escape from a state where the inverse operation can be executed.

We now propose estimator functions for the number of transitions necessary from the current state to reach a deadlock state.

**Active Processes.** In a deadlock state, all processes are blocked. The active process heuristics uses the number of active or non-blocked processes in a given state $S$:

$$H_{ap}(S) = \sum_{P_i\in\mathcal{P}\wedge active(i,S)} 1$$

where $active(i, S)$ is defined as

$$active(i, S) \equiv \bigvee_{t=(pc_i,v)\in T_i} executable(t)$$

Given that the range of $H_ap$ is $[0..|\mathcal{P}|]$, the active processes heuristic may not be very informative for protocols involving a small number of processes.

**Characterization of Deadlocks.** Deadlocks are global system states in which no progress is possible. Obviously, in a deadlock state each process is blocked in a local state that does not possess an enabled transition. It is not trivial to define a logical predicate that characterizes a state as a deadlock state which could at the same time be used

as an input to the estimation function $H_f$. We first explain what it means for a process $P_i$ to be blocked in its local state $u$. This can be expressed by the predicate *blocked*$_s$ which states that the program counter of process $P_i$ must be equal to $u$ and that no outgoing transition $t$ from state $u$ is executable.

$$\textit{blocked}_s(i,u,S) \equiv pc_i(S) = u \ \wedge \bigwedge_{t=(u,v)\in T_i} \neg \textit{executable}(t,S)$$

Suppose we are able to identify those local states in which a process $i$ can block, i.e., in which it can perform a potentially blocking operation. Let $C_i$ be the set of potentially blocking states within process $i$. A process is blocked if its control resides in some of the local states contained in $C_i$. Hence, we define a predicate for determining whether a process $P_i$ is blocked in a global state $S$ as the disjunction of *blocked*$_s(i,u,S)$ for every local state $u$ contained in $C_i$:

$$\textit{blocked(i,S)} \equiv \bigvee_{u\in C_i} \textit{blocked}_s(i,u,S)$$

Deadlocks, however, are global states in which *every* process is blocked. Hence, the disjunction of *blocked(i,S)* for every process $P_i$ yields a formula that establish wether a global state $S$ is a deadlock state or not:

$$\textit{deadlock}(S) = \bigwedge_{i=1..n} \textit{blocked}(i,S).$$

Now we address the problem of identifying those local states in which a process can block. We call these states *dangerous*. A local state is dangerous if the executabily condition of every outgoing local transition can be false. Note that some transitions are always executable, for example those corresponding to assignments. To the contrary, conditional statements and communication operations are not always executable. Consequently, a local state which has only potentially non-executable transitions should be classified as dangerous. Additionally, we allow the protocol designer to identify states as dangerous.

The deadlock characterization formula *deadlock* is constructed before the verification starts and is used during the search by applying the estimate $H_f$, with $f$ being *deadlock*. Due to the first conjunction of the formula, estimating the distance to a deadlock state is done by summing the estimated distances for blocking each process separately. This assumes that the behaviour of processes is entirely independent and obviously leads to a non-optimistic estimate. We estimate the number of transitions required for blocking a process by taking the minimum estimated distance for a process to reach a local dangerous state and negate the executability of each outgoing transition in that state. This could lead again to a non-optimistic estimate since we are assuming that the transitions performed to reach the dangerous state have no effect on disabling the outgoing transitions of that state.

It should be noted that *deadlock* characterizes many deadlock states that could be never reached by the system. Consider two processes $P_i, P_j$ having local dangerous states $u, v$, respectively. Assume that $u$ has an outgoing transition for which the executability condition is the negation of the executability condition for the outgoing transition from $v$. In this particular case it is impossible to have a deadlock in which $P_i$ is blocked in local state $u$ and $P_j$ is blocked in local state $v$, since either one of the two transitions must be executable. As a consequence the estimate could give good values to states unlikely to lead to deadlocks. Another concern is the size of the resulting formula. In

an extreme case each state of each process could be dangerous. This results in a formula of size $\prod_{i=1..n} |S_i|$. The estimate computation for this formula will be rather costly while providing a poor guide for the search algorithm. We believe that the use of the user-guided characterization of states as dangerous can be helpful to overcome this problem.

## 14.5 The HSF-SPIN Tool Set

We chose SPIN as a basis for HSF-SPIN. It inherits most of the efficiency and functionality from the original source of SPIN as well as the sophisticated search capabilities from the Heuristic Search Framework (HSF) [92]. HSF-SPIN uses a large subset of Promela as modelling language. HSF-SPIN possesses a refined state description of SPIN to incorporate solution length information, transition labels and predecessors for solution extraction. It provides an interface consisting of a node expansion function, initial and goal specification. In order to direct the search, we implemented different heuristic estimates. HSF-SPIN writes SPIN-compatible trail information that can be visualized in the XSPIN interface. As when working with SPIN, the validation of a model with HSF-SPIN is done in two phases: first the generation of an analyzer of the model, and second the validation run. The protocol analyzer is generated with the program `hsf-spin` which is a modification of the SPIN analyzer generator. By executing `hsf-spin -a <model>` several `c++` files are generated. These files are part of the source of the model checker for the given model. They are compiled and linked with the rest of the implementation, incorporating, for example, data structures, search algorithms, heuristic estimates, statistics and solution generation. HSF-SPIN also supports *bit-state hashing* by implementing Partial IDA*. HSF-SPIN can be invoked with different parameters: kind of error to be detected, property to be validated, algorithm to be applied, heuristic function to be used, weighting of the heuristic estimator. HSF-SPIN allows textual simulation to interactively traverse the state space which greatly facilitates in explaining witnesses that have been found.

HSF-SPIN is still a prototype. Therefore, its performance in terms of time and space cannot compete with SPIN. For example, an exhaustive exploration of the state space generated by the GIOP protocol parameterized with 2 clients and 2 servers is performed by SPIN (without partial order reduction) in 226 seconds with a memory consumption of 236 MB, while our tool requires 341 seconds and about 441 MB of space. Further experiments show that SPIN achieves a speedup of about factor 3 in comparison to HSF-SPIN.

## 14.6 Safety Property Validation Experiments

In this Section we present out experimental results for directed model checking of safety properties. The experiments have been performed with SPIN version 3.3.10 and HSF-SPIN version 1.0 and were executed on a SUN workstation, UltraSPARC-II CPU with 248 Mhz under Solaris 5.7. If nothing else is stated the depth bound is set to 10,000 and no compression technique is used. In the case of deadlock detection in HSF-SPIN, $H_{ap}$ is the estimation function used, unless indicated otherwise. In all other cases the formula based heuristic $H_f$ is used. When comparing to SPIN it should be noted that this model checker was invoked with partial order reduction enabled.

| GIOP | BFS | DFS | A* | BF | SPIN |
|---|---|---|---|---|---|
| s | 40,847 | 218 | 31,066 | 117 | 326 |
| e | 37,266 | 218 | 27,061 | 65 | 326 |
| t | 151,671 | 327 | 108,971 | 126 | 364 |
| l | 58 | 134 | 58 | 65 | 134 |
| **Philosophers** | **BFS** | **DFS** | **A*** | **BF** | **SPIN** |
| s | 3,678 | 1,341 | 67 | 493 | 1,341 |
| e | 2,875 | 1,341 | 17 | 225 | 1,341 |
| t | 15,775 | 1,772 | 73 | 622 | 1,772 |
| l | 34 | 1,362 | 34 | 66 | 1,362 |
| **Optical** | **BFS** | **DFS** | **A*** | **BF** | **SPIN** |
| s | 148,591 | 20 | 83 | 83 | 20 |
| e | 110,722 | 20 | 14 | 14 | 20 |
| t | 621,216 | 20 | 83 | 13 | 20 |
| l | 38 | 44 | 38 | 38 | 44 |
| **Marriers** | **BFS** | **DFS** | **A*** | **BF** | **SPIN** |
| a | 9,459 | 10,588 | 9,208 | 7,154 | 2,530 |
| e | 9,004 | 10,588 | 8,335 | 4,124 | 2,530 |
| t | 24,064 | 29,069 | 22,298 | 9,710 | 3,116 |
| l | 50 | 72 | 50 | 61 | 72 |

Table 14.3: Deadlock detection in various protocols.

### 14.6.1 Shorter Trails and Computational Effort

The first set of experiments is intended to show that A* always finds shorter trails compared to DFS while requiring less computational effort than BFS, and that in some cases A* performs better than DFS. By computational effort we mean the sum of the number of states stored, the number of states expanded and the number of transition performed. An additional objective is to show that BF can require less computational effort than A*, but that BF often delivers sub-optimal solutions.

For each kind of safety error we use a representative set of protocol models. Deadlock detection is performed using the CORBA GIOP protocol [201] with a configuration of 2 clients and 1 server, an 8-philosophers configuration of the dining philosophers problem, a model of an optical telegraph protocol [187] with 6 stations and a model of a concurrent program that solves the stable marriage problem [260] with a configuration of 3 suitors. Assertion violation detection experiments are carried out with Lynch's protocol, with a model of a relay circuit and with a faulty solution for the mutual exclusion problem (mutex)[6]. Invariant violation is evaluated using the POTS telephony model [202][7] and using an elevator protocol[8]. For the POTS model, the invariant described in Section 14.2.4 was used. In the elevator model, the invariant was of the form $\Box(\neg opened \vee stopped)$.

Tables 14.3, 14.4 and 14.5 depict the results of error detection in these protocols with various search strategies. For each protocol, the number of stored states (s), expanded states (e), transitions performed (t) and the length of the error trail (l) is shown. Similar

[6]Available from `netlib.bell-labs.com/netlib/spin`

[7]The Promela sources and further information about these models can be obtained from `www.informatik.uni- freiburg.de/ lafuente/models/models.html`

[8]Derived from `www.inf.ethz.ch/personal/biere/teaching /mctools/elsim.html`

| Relay | BFS | DFS | A* | BF | SPIN |
|---|---|---|---|---|---|
| s | 905 | 342 | 738 | 162 | 342 |
| e | 707 | 342 | 663 | 48 | 342 |
| t | 2,701 | 718 | 2,262 | 263 | 870 |
| l | 12 | 190 | 12 | 28 | 190 |
| **Lynch** | **BFS** | **DFS** | **A*** | **BF** | **SPIN** |
| s | 80 | 48 | 73 | 49 | 46 |
| e | 77 | 48 | 70 | 46 | 46 |
| t | 94 | 49 | 87 | 59 | 48 |
| l | 29 | 46 | 29 | 29 | 46 |
| **Mutex** | **BFS** | **DFS** | **A*** | **BF** | **SPIN** |
| s | 363 | 202 | 38 | 39 | 202 |
| e | 344 | 202 | 21 | 24 | 202 |
| t | 688 | 363 | 42 | 48 | 363 |
| l | 15 | 54 | 15 | 15 | 54 |

Table 14.4: Detection of assertion violations in various protocols.

| POTS | BFS | DFS | A* | BF | SPIN |
|---|---|---|---|---|---|
| s | 24,546 | - | 6,654 | 781 | 148,049 |
| e | 17,632 | - | 3,657 | 209 | 148,049 |
| t | 99,125 | - | 18,742 | 1,067 | 425,597 |
| l | 81 | - | 81 | 83 | 2,765 |
| **Elevator** | **BFS** | **DFS** | **A*** | **BF** | **SPIN** |
| s | 38,662 | 279 | 38,598 | 2,753 | 292 |
| e | 38,564 | 279 | 38,506 | 2,297 | 292 |
| t | 160,364 | 356 | 160,208 | 5,960 | 348 |
| l | 203 | 510 | 203 | 421 | 510 |

Table 14.5: Detection of invariant violations in various protocols.

to SPIN, we count a sequence of atomic steps as one unique transition. The number of expansion steps in SPIN is the number of stored states. Columns 2 to 5 correspond to different search strategies of HSF-SPIN, namely breadth-first search (BFS), depth-first search (DFS), A* and best-first search (BF). The last column corresponds to the exploration with SPIN's depth-first search (SPIN).

In all examples BFS and A* provide optimal counterexamples. Compared to BFS the A* algorithm requires less computational effort. The reduction in the number of expansions, states and transitions varies from example to example. This is mainly due to the quality of the heuristic estimate. For example, in the case of invariant violation detection for the elevator protocol, the savings in trail length achieved by A* are rather weak. This can be attributed to the integer range $[0..2]$ of the heuristic estimation function which is very small considering that the optimal solution has 203 steps. On the other hand, while detecting the violation of the invariant of the POTS protocol the heuristic function returns estimates in the range $[0..42]$. With this range, the estimate function allows for a much more differentiated successor selection in A* which results in a much more informed search leading to a strong reduction in the computational effort required to detect the error. As can be expected, DFS finds error trails significantly larger than the optimal one(s). For example, the trail provided by SPIN's DFS for the invariant violation in the

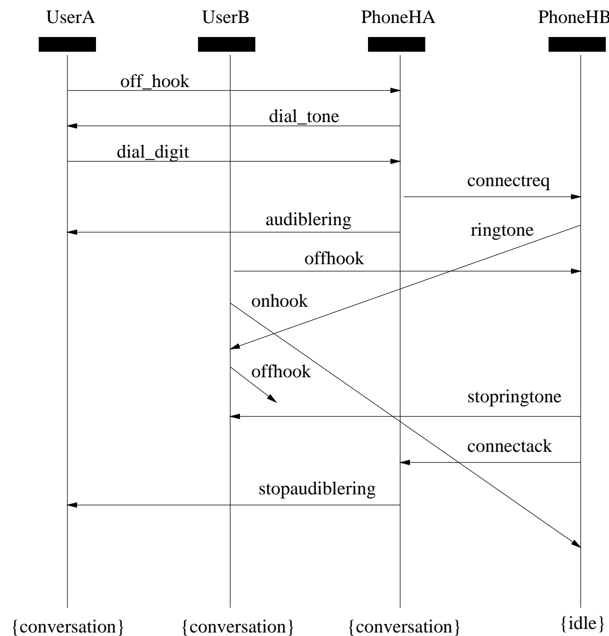


Figure 14.12: POTS example, error trail generated by HSF SPIN using A* and $H_f$.

POTS protocol is about 20 times larger than the optimal trail generated by HSF-SPIN visualized in Figure 14.12. This trail is even superior to the manually generated short trail in Figure 14.4. However, HSF-SPIN happens to find a different target state than the one found by SPIN and this target state also corresponds to a different race condition than the one found by SPIN. Nevertheless, this race condition can also be traced back to a lack of synchronization between the `UserB` and `PhoneHB` processes. While in most cases DFS performs better than A* in terms of computational effort, in the philosophers problem and in the POTS protocol the performance of A* is superior to that of DFS. The reason for this lies in the particular structure of these problems. For both problems it is necessary that there is a sequence of actions in which every process performs one or a few steps in order to get closer to the target state. DFS, however, will try to first explore all possibilities for one process, before it includes the behavior of other processes. As a consequence, DFS will require more computational effort to reach a target state than A*. It should also be explained why HSF-SPIN runs out of memory in the POTS example, while the DFS in SPIN finds a counterexample. This is due to the fact that the implementation of DFS in SPIN is more efficient, and that we employed partial order reduction. Finally, the experiments highlight that although BF often requires less computational effort than A*, the established error trails are not optimal.

## 14.6.2 Heuristic Estimates

In the previous section we have noted the important influence of the heuristic estimate function on the performance of A*. Now we analyse different heuristic functions proposed for deadlock detection. In particular we compare the heuristic based on the number of active processes $H_{ap}$ with formula based heuristic $H_f$ combined with the proposed method for automatically inferring the deadlock formula $f$. With $H_f + U$ we denote that the user explicitly defines dangerous states. In the example we chose an "optimal"

| Philosophers | no | $H_{ap}$ | $H_f$ | $H_f + U$ |
|---|---|---|---|---|
| e | 2,875 | 17 | 17 | 10 |
| r | 0..0 | 0..8 | 0..10 | 0..16 |
| Optical | no | $H_{ap}$ | $H_f$ | $H_f + U$ |
| e | 110,722 | 14 | 342 | 342 |
| r | 0..0 | 0..12 | 0..14 | 0..12 |
| Marriers | no | $H_{ap}$ | $H_f$ | $H_f + U$ |
| e | 493,840 | 432,483 | 462,235 | 192,902 |
| r | 0..0 | 0..4 | 0..25 | 0..25 |
| GIOP | no | $H_{ap}$ | $H_f$ | $H_f + U$ |
| e | 37,266 | 27,061 | 28,067 | 24,859 |
| r | 0..0 | 0..6 | 0..12 | 0..25 |

Table 14.6: Deadlock detection with A* and different heuristic functions.

labelling, i.e., exactly those states are labelled as dangerous so that the resulting global control state is a deadlock state.

In our experiments we use the deadlock solution to the philosophers problem, the optical telegraph protocol, the marriers problem and the GIOP protocol. All models have been configured as in the previous set of experiments. Table 14.6 visualizes the number of expansions required to find the deadlock state and the range of values (r) that the heuristic estimate function is defined over. In all cases the optimal solution is being found.

The results show that when applying the inferred deadlock heuristic $H_f$ user intervention improves the results in most cases. It is not easy to compare the inferred heuristic $H_f$ with $H_{ap}$. $H_{ap}$ seems to perform worse than $H_f + U$ except in the optical telegraph protocol. In the optical telegraph protocol the estimate $H_{ap}$ works well, since the number of processes in the model is quite high. In the case of the GIOP protocol and the marriers model the number of processes is rather small and $H_{ap}$ produces poor reductions in the number of expanded states. It should be emphasized that the quality of $H_f + U$ highly depends on the quality of the designers labelling of dangerous states. In summary, the experiments indicate the influence of the quality of the heuristic estimate function.

### 14.6.3 Finding Errors where DFS fails

A further objective of the *directed model checking* approach is to detect errors in models where classical depth-first exploration fails due to the exhaustion of memory resources.

We perform a set of experiments with a scalable deadlock solution to the dining philosophers problem. We let the experiments run without time limitations, but with a hard memory constraint of 512 MB. Contrary to other experiments, we allow SPIN to apply bitstate hashing compression in order to emphasize the benefits of directed search.

Table 14.7 shows results on deadlock detection in the philosophers model with a growing number of philosophers. The first column depicts the number of philosophers in the model. The labelling of the other columns is obvious.

While A* and BF seem to scale linearly with respect to the increase of $p$, BFS and DFS do not. HSF-SPIN's BFS and DFS exploration are not able to find the deadlock situation in configurations with more than 13 philosophers. SPIN can go a little further, but fails in configurations with more than 15 philosophers. The results show that there

| $p$ | | BFS | DFS | A* | BF | SPIN |
|---|---|---|---|---|---|---|
| 2 | s | 9 | 6 | 6 | 6 | 6 |
| | e | 7 | 6 | 6 | 4 | 6 |
| | t | 10 | 7 | 4 | 6 | 7 |
| | l | 10 | 10 | 10 | 10 | 10 |
| 3 | s | 19 | 19 | 12 | 26 | 19 |
| | e | 14 | 10 | 7 | 23 | 10 |
| | t | 29 | 12 | 13 | 43 | 12 |
| | l | 14 | 18 | 14 | 14 | 18 |
| 4 | s | 56 | 45 | 19 | 70 | 45 |
| | e | 42 | 45 | 9 | 57 | 45 |
| | t | 116 | 62 | 21 | 142 | 116 |
| | l | 18 | 54 | 18 | 26 | 54 |
| 8 | s | 3,768 | 1,341 | 67 | 493 | 1,341 |
| | e | 2,875 | 1,341 | 17 | 225 | 1,341 |
| | t | 15,775 | 1,772 | 73 | 622 | 1,772 |
| | l | 34 | 1,362 | 34 | 66 | 1,362 |
| 14 | s | - | - | 199 | 1,660 | 2,164,280 |
| | e | - | - | 29 | 1,963 | 2,164,280 |
| | t | - | - | 211 | 684 | 27,050,400 |
| | l | - | - | 58 | 114 | 9,998 |
| 16 | s | - | - | 259 | 2,201 | - |
| | e | - | - | 33 | 893 | - |
| | t | - | - | 273 | 2,578 | - |
| | l | - | - | 66 | 130 | - |

Table 14.7: Deadlock detection in the dining philosophers problem.

are models in which A* is able to detect errors and in which depth-first search even if combined with reduction and compression techniques fails.

### 14.6.4 IDA* and Bitstate Hashing

We now show that for given memory and time constraints, IDA* in combination with bitstate hashing is able to detect errors in problems in which A* and IDA* fail. Once the priority queue is full, A* will run out of memory and once the transposition table is full, duplicate states will force IDA* to run out of time. We use the GIOP protocol with a seeded deadlock error and configured with 3 clients and one server. We set the space limit to 256 MB and the time limit to 120 minutes. Both hash table sizes in A* and IDA* have been set to the given memory bound. Table 14.8 depicts the number of expansions performed by A*, IDA*, and IDA* combined with (double) bitstate hashing. To obtain the data in the table we modify A* to print a snapshot of the expansions every time the search depth increases, while for the last two methods, the number of state expansions corresponds to the number of nodes in the current iteration. The results show that only the combination of IDA* and bitstate hashing is able to find the error in the protocol. IDA* exceeds the time and A* exceeds the space limit.

A duplicate is a state with different generating paths. Duplicates occur frequently in typical protocols. As long as the visited lists of A* and IDA* are not full, all duplicate states are detected. When the memory bound is reached, A* aborts since it is unable to allocate further states for the open and closed lists. IDA* bypasses the problem by ex-

| Depth | A* | IDA* | IDA*+bitstate |
|---|---|---|---|
| 58 | 150,344 | 146,625 | 146,625 |
| 59 | 168,191 | 164,982 | 164,982 |
| 60 | 184,872 | 184,383 | 184,383 |
| 61 | - | 206,145 | 206,145 |
| 62 | - | - | 229,626 |
| 63 | - | - | 255,411 |
| 64 | - | - | 282,444 |
| 65 | - | - | 311,340 |
| 66 | - | - | 341,562 |
| 67 | - | - | 373,422 |
| 68 | - | - | 407,310 |
| 69 | - | - | 442,941 |
| 70 | - | - | goal found |

Table 14.8: Deadlock detection in the GIOP protocol under memory constraints.

ploring the tree expansion of the underlying graph and possibly re-exploring state space sub-tree structures. In some cases, there are too many duplicates such that after the transposition table is full and IDA* fails to complete the next iteration within the given time limit. However, IDA* with bitstate hashing prunes off duplicates optimistically, storing only a finger print (signature) of each state. This reduces the space requirements by some orders of magnitudes (about 3 in the example case), so that duplicates can be detected even in large search depths. The loss of states by false positives is marginal: in the example no state is wrongly identified with double bitstate hashing until the depth is reached in which IDA* gives up.

## 14.7 Liveness Property Validation

One feature of the Nested-DFS algorithm described in Section 14.2 is that a state, once *flagged* will not be expanded again during the cycle detection. For the correctness of the algorithm the post-order traversal of the search tree is crucial, such that the second depth-first traversal only encounters nodes that have already been visited in the main search routine. The second search can be improved by directed cycle detection search. Since we are aiming for states that have been placed on the Nested-DFS stack by the first traversal we can use heuristics to perform a directed search for the cycle-closing states. Unfortunately, in each of our benchmark examples, there is at most one accepting cycle, so that there is nothing to improve. The disadvantage of a pre-ordered nested search approach (search the acceptance state in the never claim and, once encountered, search for a cycle) is its quadratic worst-case time and linear memory overhead, since the second search has to be invoked with a newly initialised list of flagged states. To address this drawback, we developed an improvement of the nested depth-first search algorithm that exploits the structure of the never claim. This algorithm is applicable to a large set of practical property specifications, and can be combined with heuristic techniques for more efficient search performance.

### 14.7.1 Classification of Never Claims

Strongly connected components (SCC) partition a directed graph into groups such that there is no cycle combining two components. A subset of nodes in a directed graph is strongly connected if for all nodes $u$ and $v$ there is a path from $u$ to $v$ and a path from $v$ to $u$. SCCs are maximal in this sense and can be computed in linear time by applying Tarjan's algorithm [70].

To illustrate how SCCs can help improve the Nested-DFS algorithm, consider the never claim of Figure 14.1. We find two strongly connected components: the first is formed by $n_0$ and the second by $n_a$. Furthermore, there is no path from the second SCC to the first. Accepting cycles in the synchronous product automaton are composed of states in which the never claim is always in the second SCC. We can conclude that if the local never claim state corresponding to a cycle-closing synchronous product automaton state belongs to the second SCC the cycle is accepting. If, however, it belongs to the first SCC, it is not accepting.

In order to generalize this observation suppose that we have pre-computed all SCCs of a given never claim. Due to the synchronicity of the product of the model automaton and the never claim a cycle in the synchronous product is generated by a cycle in exactly one SCC. If the cycle is accepting, so is the corresponding cycle in the SCC of the never claim. Suppose that each SCC is either composed only of non-accepting states or only of accepting states. Then global accepting cycles only contain accepting states, while non-accepting cycles only contain non-accepting states. Therefore, a single depth-first search can be used to detect accepting cycles: if a state $s$ is found in the stack, then the established cycle is accepting if and only if $s$ itself is accepting.

The partitioning rules for SCCs given above can be relaxed according to the following classification of SCCs:

- We call an SCC *accepting* if at least one of its states is accepting, and *non-accepting* (N) otherwise.
- We call an accepting SCC *fully accepting* (F) if all of its cycles contain at least one accepting state.
- We call an accepting SCC *partially accepting* (P) if there is at least one cycle that does not contain an accepting state.

If the never claim contains no partially accepting SCC, then accepting cycle detection for the global state space can be performed by a single depth-first search: if a state is found in the stack, then it is accepting, if the never state belong to an accepting SCC. A special case occurs if the never claim has an endstate. When this state is reached, the never claim is said to be violated and a *bad* sequence has been found. Bad sequences are tackled similarly to safety properties by standard heuristic search.

The classification of patterns in property specifications proposed in [87] reveals that an empirically collected database of 555 practically used LTL properties partitions into *Absence (A)* (85/555), *Universality (U)* (119/555), *Existence (E)* (27/555), *Response (R)* (245/555), *Precedence (P)* (26/555), and *Others* (53/555). Using this pattern scheme and the scope modifiers *Globally (G), Before (B), After (A), Between (B)*, and *Until (U)* we obtain a partitioning into SSCs according to Table 14.9. We indicate the presence of end-states with the letter $S$. Since the specification patterns are given using LTL formulae,

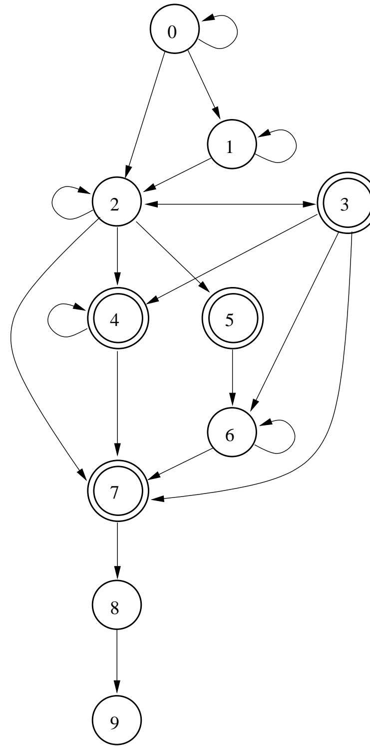


Figure 14.13: Never Claim for a response between property.

| | G | B | A | B | U |
|---|---|---|---|---|---|
| A | S,N | S,N | S,N | S,N | S,N,P |
| U | S,N | S,N | S,N | S,N,P,F | S,N,P |
| E | F | S,P,N | N,F | S,N,P | S,N,F |
| R | N,F | S,N,P,F | N,F | S,N,P,F | S,N,P,F |
| P | S,N,P | S,N | N,P | S,N | S,N,P |

Table 14.9: SCC classification for LTL specification patterns.

we derive the equivalent never claims using the SPIN built-in LTL to never claim converter. Then, we apply an algorithm that computes the SCCs of the state transition graph of the never claim automaton and that classifies them into the different classes. For example, Figure 14.13 depicts the state transition graph of the never claim corresponding to a *response pattern* with *between* scope for which the corresponding LTL formula is $\Box((q \wedge \neg r \wedge \diamond r) \rightarrow (p \rightarrow (\neg r U(s \wedge \neg r)))Ur)$. The graph is classified as follows: SCCs $\{0\}$, $\{1\}$ and $\{6\}$ are of class N, $\{2,3\}$ is of class P and $\{4\}$ of class F. State 9 is and endstate and the rest of the states are *transient* states.

Our approach is particularly useful for never claims that only contain N and F components, as for instance the Response pattern with a global scope. Given the prevalence of the Response pattern we conclude that our improvement of the Nested-DFS algorithm is applicable to a large set of practical problems.

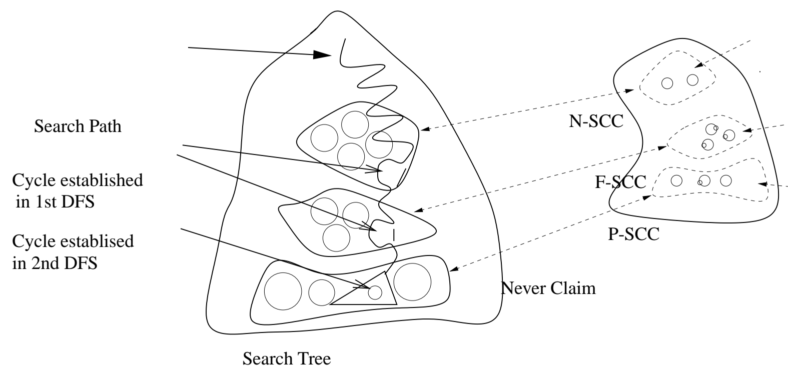


Figure 14.14: Visualization of the different cases in Improved-Nested-DFS.

### 14.7.2 Improved Nested Depth-First Search

We now present the *Improved-Nested-DFS* (INDFS) algorithm that is based on the above ideas, given in Figure 14.15. In this Figure, SCC$(s)$ is the SCC of state $s$, F-SCC$(s)$ determines if the SCC of state $s$ is of type F (fully accepting), P-SCC$(s)$ determines if the SCC of the state is of type P (partially accepting) and *neverstate*$(s)$ denotes the local state of the never claim in the global state $s$.

The algorithm finds acceptance cycles without nested search for all problems which partition into N- or F-components. Except for P-SCCs it avoids the post-order traversal. For P-SCCs we guarantee that the second cycle detection traversal is restricted to the strongly connected component of the seed. The algorithm considers the successors of a node in depth-first manner and marks all visited nodes with the label *hash*. If a successor $s'$ is already contained in the stack, a cycle $C$ is found. If $C$ corresponds to a cycle in a F-SCC of the *neverstate* of $s'$, it is an accepting one. Cycles for the P-SCCs parts in the never claim are found as in Nested-DFS, with the exception that the successors of a node are pruned which *neverstates* are outside the component. If an endstate in the never claim is reached the algorithm terminates immediately. A detailed proof of the correctness of INDFS is given in Section 14.7.4.

Figure 14.14 depicts the different cases of cycles detected in the search. The main idea for the correctness of Improved-Nested-DFS is based on the fact that all cycles in the state-transition graphs correspond to cycles in the never claim. Therefore, if there is no cycle combining two components in the never claim, so there is none in the overall search space.

As mentioned above, the strongly connected components can be computed in time linear to the size of the Never Claim, a number which is very small in practice. Partitioning the SCCs in *non-accepting*, *partially accepting* and *fully accepting* can also be achieved in linear time by a variant of Nested-DFS in the never claim.

**Improved-Nested-DFS**$(s)$
  $hash(s)$
  **for all** successors $s'$ of $s$ **do**
    **if** $s'$ in *Improved-Nested-DFS*-Stack **and**
      *F-SCC*$(neverstate(s'))$ **then**
        **exit***LTL-Property violated*
    **if** $s'$ not in the hash table **then Improved-Nested-DFS**$(s')$
  **if** $accept(s)$ **and** *P-SCC*$(neverstate(s))$ **then**
    **Improved-Detect-Cycle**$(s)$

**Improved-Detect-Cycle**$(s)$
  $flag(s)$
  **for all** successors $s'$ of $s$ **do**
    **if** $s'$ on $Improved-Nested-DFS$-Stack **then**
      **exit** *LTL-Property violated*
    **else if** $s'$ not flagged **and**
      $SCC(neverstate(s)) = SCC(neverstate(s'))$ **then**
        **Improved-Detect-Cycle**$(s')$

Figure 14.15: Improved Nested Depth-First Search.

### 14.7.3 A* and Improved-Nested-DFS

So far we have not considered heuristic search for Improved-Nested-DFS. Once more, we consider the example of *Response* properties to be validated. In a first phase, states are explored by A*. The heuristic estimation function can easily be designed to reach the accepting cycles in the SCCs faster, since all states that we are aiming at are accepting. This approach generalizes to a hybrid algorithm A* and Improved-Nested-DFS, *A*+INDFS* for short, that alternates between heuristic search in N-SCCs, single-pass searches in F-SCCs, and Nested-Search in P-SCCs. If a P- or S-component is encounterd, Improved Nested-DFS is invoked and searches for cycles. The heuristic estimate respects the combination of all F-SCCs and P-SCCs, since accepting cycles are present in either of the two components. The nodes at the horizon of a F- and P-component lead to pruning of the sub-searches and are inserted back into the *Open*-List of A*, which contains all horizon nodes with a neverstate in the corresponding N-SCCs. Therefore *A*+INDFS* continues with directed search, if cycle detection in the F- and P-components fails. Cycle detection search itself can be accelerated with an estimation function heading back to the states where it was started.

Figure 14.16 visualizes this strategy for a response property. The never claim has the following SCCs: $SCC_0$ which is a N-SCC, and $SCC_a$ which is F-SCC. The state space can be seen as divided in two partitions, each one composed of states where the never claim is a state belonging to one of the SCCs. In a first phase, A* is used for directing the search to states of the partition corresponding to $SCC_a$. Once a goal state is found, the second phase begins, where the search for accepting cycles is performed by Improved-Nested-DFS.

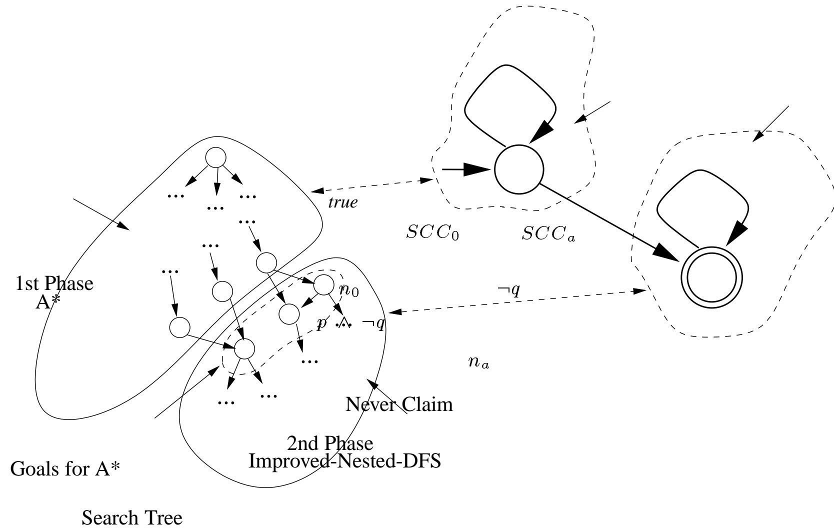


Figure 14.16: Visualization of A* and Improved-Nested-DFS for a response property.

### 14.7.4 Correctness of INDFS

The nested depth-first search algorithm in model checking validates the emptiness of Büchi automata. It searches accepting cycles in the problem graph that represents the state-space of a Büchi automaton and reports non-emptiness if and only if there exists at least one accepting cycle in the graph. A correctness proof of this algorithm can be found in [65]. Our improvement to the nested depth-first search algorithm is depicted in Figure 14.15. To prove the correctness of the algorithm we start with some definitions.

**Definition 8** *A Büchi automaton is a five tuple $\langle \Sigma, Q, \delta, Q_0, F \rangle$, where $\Sigma$ is a finite alphabet, $Q$ is the finite set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q_0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states.*

**Definition 9** *A* run *of a Büchi automaton over an infinite word $v \in \Sigma^*$ is a mapping $\rho : \{0, 1, .., \infty\} \mapsto Q$ such that a) the first state is an initial state, that is, $\rho(0) \in Q_0$, and b) moving from the $i$-th state $\rho(i)$ to the $(i+1)$-st state $\rho(i+1)$ upon reading the $i$-th input letter $v(i)$ is consistent with the transition relation, that is, for all $i \geq 0$ we have $(\rho(i), v(i), \rho(i)+1) \in \delta$.*

**Definition 10** *Let* $\inf(\rho)$ *be the set of states that appear infinitely often in a run $\rho$ (when treating the run as an infinite path). A run $\rho$ of a Büchi automaton $B$ over an infinite word is said to be* accepting *if and only if some accepting state appears infinitely often in $\rho$, that is, when* $\inf(\rho) \cap F \neq \emptyset$. *The* language $\mathcal{L}(B)$ accepted by the Büchi automaton *$B$ is then the set of infinite words, over which all runs of $B$ are accepting.*

Let $M$ be a finite state automaton representing the model, and let $N$ be the *never claim*. For this construction the automaton $M$ is interpreted as a Büchi automaton in which *all* states are accepting.

**Definition 11** *Let $M = \langle \Sigma, Q_M, \delta_M, Q_0^M, F_M \rangle$ be a Büchi automaton which states are all accepting, that is $Q_M = F_M$, and let $N = \langle \Sigma, Q_N, \delta_N, Q_0^N, F_N \rangle$ be another Büchi automaton. The* synchronous product *$M \otimes N$ of $M$ and $N$ is defined as: $M \otimes N = \langle \Sigma, Q, \delta, Q_0, F \rangle$, where $Q = Q_M \times Q_N$, $Q_0 = Q_0^M \times Q_0^N$, $F = F_M \times F_N = Q_M \times F_N$, and $((s_M, s_N), a, (s'_M, s'_N)) \in \delta$ if and only if $(s_M, a, s'_M) \in \delta_M$ and $(s_N, a, s'_N) \in \delta_N$.*

Büchi automata can be represented as directed graphs: the set of vertices is $Q$ and the edges are labelled by the transition relation $\delta$. Runs of the automaton over an infinite word correspond to infinite paths in the graph, and accepting runs to infinite paths containing infinite accepting cycles. An accepting cycle is defined as a cycle in which at least one state is accepting.

**Definition 12** *A strongly connected component (SCC) of a directed graph is a maximal set of vertices, such that each vertex in the set is reachable from each other vertex of the set in 1 or more steps*[9].

It is not difficult to show that pairwise reachability is an equivalence relation such that the set of nodes can be partitioned into equivalence classes of strongly connected components. An important consequence of the definition of SCCs is that all vertices of a cycle belong to the same SCC. In the following we write $scc(s)$ to denote the SCC to which a state $s$ belongs.

Let $Q$ be the set of states of $M \otimes N$. We define a partition function $\pi$ from $Q$ onto $\{0, \ldots, k\}$ in such a manner that two states belong to the same partition if and only if the state component of $N$ belongs to the same SCC in the state transition graph of $N$. More precisely, if $s = (s_M, s_N)$ and $i = scc(s_N)$ then $\pi((s_M, s_N)) = i$. Obviously, $\pi$ defines a partition of equivalence classes $P_0, \ldots P_k$ of $Q$, where $P_i = \{s \in Q \mid \pi(s) = i\}$, $i \in \{0, \ldots, k\}$.

**Definition 13** *A strongly connected component is called* non-accepting *if none of its states is accepting,* full-accepting *each cycle formed by states of the SCC is accepting, and* partial-accepting *otherwise.*

**Definition 14** *An equivalence class $P_i$ of $M \otimes N$ is* non-accepting, full-accepting *or* partial-accepting *if the corresponding strongly connected component $i$ in $N$ is* non-accepting, full-accepting *or* partial-accepting*, respectively.*

**Lemma 9** *If there is a cycle $C$ in $Q$, then $\pi$ partitions the states in $Q$ in such a manner that all states of the cycle belong to the same equivalence class in $Q$, i.e., $C \subseteq P_i$ for one $i \in \{0, \ldots, k\}$.*

**Proof:** Let $C$ be a cycle in state transition graph of $Q$, that is $C = (s_0, s_1, \ldots, s_n)$ with $s_n = s_0$ and $(s_i, a, s_{i+1}) \in \delta$ for all $i \in \{0, \ldots, n-1\}$. Therefore, since $s_i = (s_i^M, s_i^N)$ and $s_i^N \in N$, $i \in \{0, \ldots, n\}$, a cycle $C_N = (s_0^N, s_1^N, .., s_n^N = s_0^N)$ exists with $(s_i^N, a, s_{i+1}^N) \in \delta_N$ for all $i \in \{0, \ldots, n\}$. Hence, for all $s_i = (s_i^M, s_i^N)$ and $s_j = (s_j^M, s_j^N)$ in $C$ we have $scc(s_i^N) = scc(s_j^N)$. This implies $\pi(s_i) = \pi(s_j)$ for all $s_i, s_j \in C$ such that all states of $C$ belong to the same equivalence class. □ ■

[9] Requiring reachability in 1 or more steps is not the standard definition of an SCC. However, the minimum path length of 1 is necessary for a concise proof of our algorithm

**Lemma 10** *A cycle $C$ in $M \otimes N$ is accepting if and only if the corresponding cycle in $N$ is accepting.*

This is easy to see, since as defined, a state $s = (s_M, s_N)$ of $M \otimes N$ is accepting if and only if $s_N$ is an accepting state of $N$.

**Lemma 11** *All cycles in a* non-accepting *component are non-accepting, all cycles in a* fully-accepting *component are accepting. In* partial-accepting *components, there can be accepting and non-accepting cycles.*

Lemma 11 is immediately deduced from Definitions 13 and 14, and Lemma 10.

The following lemma is a well-known property of depth-first search and is essential for proving the correctness of our algorithm.

**Lemma 12** *Let $s$ be a vertex that does not appear on any cycle. Then the depth first search algorithm will backtrack from $s$ only after all the nodes that are reachable from $s$ have been explored and backtracked from.*

It is easy to see that this lemma still holds for the first search in both the original and in the improved nested depth first search algorithm.

**Theorem 16** *The improved nested depth first search algorithm returns a counterexample for the emptiness of the checked automaton $M \otimes N$ exactly when $\mathcal{L}(M \otimes N)$ is not empty.*

**Proof:** We have to show $I$) that a counterexample returned by the algorithm corresponds in fact to an accepting run of the automaton, and $II$) that no accepting run is missed by the algorithm.

$I$) The first thing to show is that if the algorithm finds an accepting cycle, then the cycle is in fact accepting. The algorithm closes accepting cycles in the first and in the second search. When the algorithm closes cycles in the second search, it acts like the original algorithm. As shown in [65], cycles closed in the second search are accepting, since the second search is started from accepting states only. On the other hand cycles closed in the first search correspond only to cycles present in a *full-accepting* equivalence class, and as shown in Lemma 10, every cycle in a *full-accepting* component is accepting.

$II$) The difficult case is to prove that if the algorithm finds no accepting cycle then $\mathcal{L}(M \otimes N)$ is in fact empty. As shown above, accepting cycles can exist only in *full-accepting* component or in *partial-accepting* component. There are two cases if the algorithm fails to find an existing accepting cycle: $II_a$) the cycle exists in a *full-accepting* component and is missed in the first search, or $II_b$) the cycle exists in a *partial-accepting* component and is missed in the second search.

$II_a$) Suppose that an accepting cycle exists in a *full-accepting* component and that the first search fails to find it. Let $s$ be the first state visited by the depth first search that is reachable from itself and that belongs to a *full-accepting* component. The first search misses that cycle if in the moment in which the search is started from $s$, every path from $s$ to itself contains a an already visited state. Let $s'$ be the first such state. Then $s'$ was visited by the depth-first search before $s$ and is reachable from itself through the cycle $(s' \to \ldots \to s \to \ldots \to s')$, and $s'$ belong to the same *full-accepting* component by Lemma 9, which contradicts our assumption.

$II_b$) Suppose now that an accepting cycle exists in a *partial-accepting* component and that the second search fails to find it. In this case a similar reasoning as in [65] can be done to show that this cannot be happen. Let $s$ be the first accepting state belonging to a *partial-accepting* component from which the second search starts but fails to find a cycle even though one exists. In the moment in which the second search starts from $s$ there is at least one flagged state on a cycle through $s$. Let $r$ be the first such state, and let $s'$ be the state from which the second search that flagged $r$ was started. In the algorithm the second search remains in the same equivalence class from which the search was started. Therefore $s$, $r$ and $s'$ must belong to the same component. According to our assumptions, the second search from $s'$ was started before the second search from $s$. There are two cases:

$II_b'$: The state $s'$ is reachable from $s$. Then there is a cycle $(s' \rightarrow \ldots \rightarrow r \rightarrow \ldots \rightarrow s \rightarrow \ldots \rightarrow s')$ that could not have been found previously, otherwise the algorithm would already have terminated. By Lemma 11, the cycle belongs to a *partial-accepting* component. However this contradicts our assumption from which the second search missed a cycle belonging to a *partial-accepting* component.

$II_b''$: The state $s'$ is not reachable from $s$. If $s'$ appears on a cycle, then a cycle was missed before starting the second search at $s$ and the cycle belongs to a *partial-accepting* component, since $s$ and $s'$ belong to the same component. According to the assumption, $s$ is reachable from $r$ and, subsequently, $s$ is reachable from $s'$. Thus, if $s'$ does not occur on a cycle, by Lemma 12 we must have discovered and backtracked from $s$ in the first search before backtracking from $s'$. Hence, according to the algorithm, we must have started a second search from $s$ before starting it from $s'$. This contradicts the assumptions. □ ■

## 14.8 Liveness Property Experiments

In this Section we describe experimental results in the validation of liveness properties. The experimental setup is largely as described in Section 14.6. We compare two algorithms in this section, namely NDFS and INDFS. Both algorithms have been implemented in HSF-SPIN. We have also implemented INDFS inside SPIN, so that both tools have the same algorithmic capabilities. Since the results produced by both tools are very similar in terms of computational effort, we only give the values obtained by HSF-SPIN in this Section.

### 14.8.1 INDFS for Validating Correctness

This first set of experiments is intended to show the benefits of INDFS when validating liveness properties. In the worst case, INDFS performs as many transitions and expansions as NDFS, while in a best case situation INDFS can halve these values. The worst case occurs when the never claim contains no F-SCCs, while the best case occurs when the never claim contains exclusively SCCs of this type. Note that all never claims are generated using SPIN's LTL-to-never-claim translation. We use a model of the leader election algorithm as test case. As a worst case we check the property $\Diamond\Box oneLeader$ for which the corresponding never claim is formed by a unique P-SCC. For the best case situation we used the property $\Diamond elected$ for which the corresponding never claim is formed

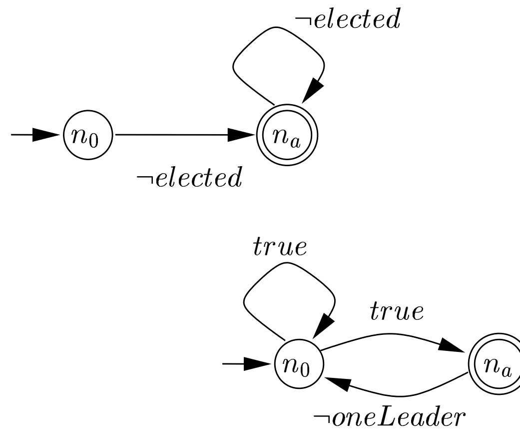


Figure 14.17: Never claims for $\Diamond\Box oneLeader$ (bottom-right) and $\Diamond elected$ (top-left).

| $\Diamond\Box oneLeader$ | NDFS | INDFS |
|---|---|---|
| s | 4,779 | 4,779 |
| e | 9,556 | 9,556 |
| t | 42,307 | 42,307 |
| $\Diamond elected$ | NDFS | INDFS |
| s | 2,380 | 2,380 |
| e | 14,086 | 7,044 |
| t | 4,759 | 2,380 |

Table 14.10: Checking correctness of two liveness properties in the leader election algorithm with NDFS and INDFS.

by a unique F-SCC. Figure 14.17 illustrates the never claims that SPIN generates for each property.

Table 14.10 depicts the results of the experiments. The number of transitions and expansions is shown. The number of stored states is also included in the table to show that both algorithms explore exactly the same number of states. The results show that in the worst case situation both NDFS and INDFS perform the same. On the other hand, in the best case situation INDFS requires about half of the transitions and expansions that NDFS requires.

### 14.8.2 INDFS for Error Detection

Our objective now is to show that INDFS requires less computational effort and provides better error trails than NDFS. We also test the performance of the hybrid algorithm that combines NDFS with A*.

We first use a version of the GIOP model configured with 1 server and 3 clients and with a seeded error that causes the model to violates a response property stating that when a client sends a request, a reply will always be received. Second, we use a model of an elevator with 3 floors that violates the response property stating that whenever a request for the elevator exists in one level, the elevator will eventually stop at that level and open the door. Table 14.11 shows experimental results on detecting the violation of LTL formulae. NDFS is compared with INDFS and the algorithm that combines A* and

| Elevator | NDFS | INDFS | A*+INDFS |
|---|---|---|---|
| s | 192 | 187 | 29,407 |
| e | 229 | 187 | 28,309 |
| t | 280 | 215 | 130,211 |
| l | 320 | 311 | 297 |
| GIOP | NDFS | INDFS | A*+INDFS |
| s | 7,331 | 7,260 | 86 |
| e | 7,346 | 7,260 | 81 |
| t | 33,061 | 32,984 | 93 |
| l | 289 | 155 | 155 |

Table 14.11: Detection of violation of liveness properties in two protocols.

NDFS (A*+INDFS).

The results show that INDFS provides small improvements over NDFS in all categories. However, only for the GIOP protocol the reduction is significant, INDFS almost halves the length of the error trail. The hybrid algorithm finds better solutions in all situations, but its computational effort varies drastically. While in the elevator experiment it requires about 15 times more state expansions than INDFS, in the GIOP experiment it performs 89 times less. The reason of this varying behavior is that A*+NDFS directs the search to the nearest full accepting component of the state space. This component may, however, be free of cycles. Only after this part of the state space is entirely explored the nested search returns control to A* which then directs the search into the next full accepting part. While in the case of the GIOP protocol the algorithm finds a component with a cycle early on in the search, in the elevator example the algorithm first explores parts of the state space that include accepting states, but no accepting cycles.

## 14.9 Related Work

In earlier work on the use of directed search in model checking the authors apply best-first exploration to protocol validation [239]. They are interested in typical safety properties of protocols, namely unspecified reception, absence of deadlock and absence of channel overflow. In the heuristics they use an estimate determined by identifying *send* and *receive* operations. In the analysis of the X.21 protocol they obtained savings in the number of expansion steps of about a factor of 30 in comparison with a typical depth first search strategy. We have incorporated this strategy in HSF-SPIN. The approach in [239] is limited to the detection of deadlocks, channel overflows and unspecified reception in protocols with asynchronous communication. To the contrary our approach is more general and handles a larger range of errors and communication types. While the measures in [239] are merely stochastic and will not yield optimal solutions, the heuristics we propose are in most cases lower bound estimators and hence allow us to find optimal solutions.

Recent work [152] applies heuristic search to the verification of java programs. It is proposing heuristics that increase coverage of the program while disregarding a targeted search for error states. This approach does not guarantee optimal counterexamples and accomplishes faster error finding through improved code coverage.

The same holds for recent work [147] that proposes the application of genetic algorithms for finding errors in very large state spaces. Genetic algorithms require *fitness*

functions which are a variant of heuristic evaluation functions. Different heuristics for deadlock detection and assertion violation based on enabledness of transitions and message exchanges are proposed.

The identification of three phases in the verification process is at the heart of work documented in [68]. In *exploratory mode* the system designer tries to find a first error, in *fault-finding mode* s/he aims at meaningful counterexamples, while in the *maintenance mode* one does not expect errors at all. From this point of view, our approach concentrates on the first two modes. Moreover, the authors in [68] analyse which algorithm is best-suited for which mode. They use different variants of depth-first search, breadth-first search and A*. Some of the ideas for the heuristic estimates are similar to ours, but the authors do not elaborate on the specific heuristic estimates that they use. Contrary to us, they do not consider IDA* and restrict their work to safety properties. In comparison, cur conclusions are slightly different from theirs. We agree that a shortest path algorithm is suitable for the fault-finding mode, but we believe that directed search can also be useful in the first exploratory mode: even in this phase by guiding the search an error state can be found with less computational effort than with blind search strategies.

The authors of [352] use BDD-based symbolic search within the Mur$\phi$ validation tool. The best first search procedure they propose incorporates symbolic information based on the Hamming distance of two states. This approach has been improved in [303], where a BDD-based version of the A* algorithm [112] for the $\mu$cke model checker [35] is presented. The algorithm outperforms symbolic breadth-first search exploration for two scalable hardware circuits. The heuristic is determined in a static analysis prior to the search taking the actual circuit layout and the failure formula into account. The approach to symbolic guided search in CTL model checking documented in [38] applies `hints' to avoid sections of the search space that are difficult to represent for BDDs. This permits splitting the fix-point iteration process used in symbolic exploration into two parts yielding under- and over-approximation of the transition relation, respectively. Benefits of this approach are simplification of the transition relation, avoidance of BDD blow-up and a reduced amount of exploration for complicated systems. However, in contrast to our approach providing `hints' requires user intervention. Also, this approach is not directly applicable to explicit state model checking, which is our focus.

The need for heuristics is apparent in conformant planning, where the symbolic representation compensates partial knowledge of the current state. The work of [31] trades information gain for exploration time with an estimate preferring belief states with low cardinality.

Timed automata call for a finite partitioning of the state space through a symbolic representation of states as a reduced set of difference constraints. One example is the real-time model checker Uppaal, which has also been accelerated by heuristic search to optimise different cost-functions with A* [25]. The techniques are reported to reduce the explored state-space with up to 90%.

Exploiting structural properties of the Büchi Automaton in explicit state mode checking has been considered in the literature in the context of weak alternating automata (WAA) [267]. WAA were invented to reason about temporal logics, generalize the transition function with boolean expressions of the successor set, and partition the automaton structure. The classification of the states of a WAA differs from ours, since the partitioning into disjoint sets of states that are either all accepting or all rejecting does not imply our partitioning. The simplification of Büchi automata proposed in [330] is inferred from

an LTL property, whereas our INDFs algorithm is based on the analysis of the structure of Büchi automata. The work in [330] also considers a partitioning according to WAA-type weakness conditions and hence differs from the approach taken in our paper.

The approach taken in [345] addresses explicit CTL* model checking in SPIN using hesitant alternating automata (HAAs). The paper shows that the performance of the proposed `LTL non-emptiness game' is in fact a reformulation and improvement of nested depth-first search. Both the partitioning and the context of HAA model checking are significantly different from our setting.

## 14.10 Conclusion and Outlook

We argued that in order to facilitate debugging the error trails or witnesses that a model checker generates minimizing their length is highly desirable. A reduction in the number of visited states during state space search is also desirable since this renders larger models tractable. Standard depth-first search algorithms used in explicit state model checkers like SPIN are rather efficient in terms of memory usage and computing time, but tend to produce lengthy counterexamples.

We introduced into heuristic search algorithms, and showed how to apply heuristic search to safety property validation. The experimental results showed that directed model checking with A* always returns shorter error trails than DFS, and that in most instances the trail length is optimal. Regarding computational effort the results were mixed: in some instances A* was superior to SPIN and DFS, but in many cases A* was not performing as well. It also became clear that the gain obtained through directed model checking is better the more differentiation the heuristic estimation function allows. We also observed that for the dining philosophers problem under constrained memory availability directed model checking was able to solve a problem that could not be solved by DFS. We expect that this effect is linked to the highly symmetric nature of the problem, and the high degree of coordination that is typical for this example.

Next we proposed an improvement of the nested depth-first search algorithm that exploits the structure of the never claim to be validated. The INDFS algorithms is applicable to the validation of liveness properties. We showed that INDFS, which is not a directed model checking algorithm, leads to modest improvements in terms of error trail length compared to NDFS. In further experiments we showed how the combined usage of A* and NDFS can lead to significant reductions in error trail length.

The incorporation of heuristic search strategies is based on the observation that standard state space exploration algorithms perform a search that is rather uninformed of the structure of the search problem. As raw as the heuristics that we propose may be, it is surprising to see them work rather well on many practical problems. We are not primarily interested in optimal solutions, which is why we can tolerate non-admissible heuristic estimates when optimistic estimates are not available.

In concurrent work [110] we describe an approach to shorten existing error trails using refined state distance metrics as heuristic estimates. This approach has already been implemented in HSF-SPIN. For selected benchmark and industrial communication protocols experimental evidence is given that *trail-directed model checking* effectively shortcuts existing witness paths.

We are nevertheless interested in improving the quality of the heuristics so that

our approach becomes applicable to an even larger set of problems. One approach assesses the fact that our assumption of independence in combining sub-formulae is rarely fulfilled in practice. The main idea in the not yet implemented approach of *directed stochastic model checking* is to derive a stochastic model for search prediction that takes correlations of propositions into consideration in order to direct the search.

Further work [241] investigates the combination of partial order reduction techniques with the directed model checking approach of HSF-SPIN. Both theoretically and empirically we show that A* and IDA* can be combined with partial order reduction methods. While the benefit of the application of partial order reduction to A* is limited, due to its similarity to DFS IDA* avails itself rather nicely to partial order reduction.

# Paper 15

# Trail-Directed Model Checking

Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: {edelkamp,lafuente,leue}@informatik.uni-freiburg.de

Electronic Notes in Theoretical Computer Science, Vol. 55, No 3, pp 1–16. Elsevier.

**Abstract**

HSF-SPIN is a Promela model checker based on heuristic search strategies. It utilizes heuristic estimates in order to direct the search for finding software bugs in concurrent systems. As a consequence, HSF-SPIN is able to find shorter trails than blind depth-first search.

This paper contributes an extension to the paradigm of *directed model checking* to shorten already established unacceptable long error trails. This approach has been implemented in HSF-SPIN. For selected benchmark and industrial communication protocols experimental evidence is given that *trail-directed model checking* effectively shortcuts existing witness paths.

## 15.1 Introduction

The formal methods of model checking [65] have various applications in software verification [28]. Through the exploration of large state-spaces model checking produces either a formal proof for the desired property or a detailed description of an error trail. We concentrate on explicit state model checking and its application to the validation of communication protocols.

In the broad spectrum of techniques for tackling the huge state spaces that are generated in concurrent systems, heuristic search is one of the new promising approaches for failure detection. Early precursors execute explicit best-first exploration in protocol validation [239] and symbolic best-first search in the model checker Mur$\phi$ [352]. Symbolic guided search in CTL model checking is pursued in [38] and bypasses intense symbolic computations by so-called hints. Last but not least, the successful commercial UPPAAL verifier for real-time systems represented as timed automata has also been effectively enriched by directed search techniques [24].

Our own contributions to *directed model checking* integrate heuristic estimates and search algorithms to the $\mu$cke model checker [303], to a domain independent AI-planner [93], and to a Promela model checker [109, 107]. The global state space is interpreted as an implicitly given graph spanned by a successor generator function, in which paths corresponding to error behaviors are searched. The length of the witness path is crucial to the designer/programmer to debug the erronous piece of software; shorter trails are easier to interpret in general.

In the model checker SPIN [187] safety properties are checked through a simple depth-first search of the system's state space, while liveness properties require a two-fold nested depth-first search. The error trail in the first case is a simple path from the start state to an error state, while in the second case we have a seeded cycle, that is a path composed by a prefix that leads to a seed state, followed by a cycle that is closed at this state.

Our experimental tool HSF-SPIN [1] provides AI heuristic search strategies like A*, IDA* and best-first for finding safety errors [109], and an improved version of nested depth-first search [107], based on exploting the never-claim representation of the required temporal property to simplify the checking process.

In this paper we concentrate on error trail improvement, an apparent need in practical software development. We expect that a possibly long witness for an error is already given. This trail might be found by simulation, test, random walk, or depth-first model checking. This path is read as an additional input, reproduced in the model and then significantly improved by directed search.

HSF-SPIN tries to find errors faster than traditional tools by employing heuristic search strategies for non-exhaustive, guided state space exploration. While HSF-SPIN can be used for full verification through exhaustive state space search, this is not its primary objective and we note that other model checkers, like Spin or SMV, are likely to be more time and space efficient for this purpose.

The paper is structured as follows. First we give some background on the AI technique we use. In a next section. In the next section we introduce the HSF-SPIN model checker and its usage in terms of its command line options. In the following sections we

[1] Available from
`http://www.informatik.uni-freiburg.de/~lafuente/hsf-spin`

address the facets of trail-directed search, based on the hamming distance and the FSM distance estimates. We distinguish between single-state trail directed search for safety errors and cycle-detection trail-directed search for liveness errors. Both approaches have been implemented in HSF-SPIN and in the experimental section we present first results. We close with some concluding remarks.

## 15.2 Heuristic Search

Depth-first search and breadth-first search are call *blind* search strategies, since they use no information of the concrete state space they explore. On the other hand, heuristic search algorithms take additional search information in form of an evaluation function into account. This function is used to rank the desirability of expanding a node $u$.

A* [161] uses an evaluation function $f(u)$ that is the sum of the generating path length $g(u)$ and the estimated cost of the cheapest path $h(u)$ to the goal. Hence $f(u)$ denotes the estimated cost of the cheapest solution through $u$. If $h(u)$ is a lower bound then A* is optimal, i.e. it finds solution paths of optimal length.

Table 15.1 depicts the implementation of A*, where $g(u)$ is the length of the traversed path to $u$ and $h(u)$ is the estimate distance from $u$ to a failure state.

**A***$(s)$
    *Open* $\leftarrow \{(s, h(s))\}$; *Closed* $\leftarrow \{\}$
    **while** (*Open* $\neq \emptyset$)
        $u \leftarrow$ *Deletemin*(*Open*); *Insert*(*Closed*,$u$)
        **if** (*failure*$(u)$) **exit** *Goal Found*
        **for all** $v$ **in** $\Gamma(u)$
            $f'(v) \leftarrow f(u) + 1 + h(v) - h(u)$
            **if** (*Search*(*Open*, $v$))
                **if** $(f'(v) < f(v))$
                    *DecreaseKey*(*Open*, $(v, f'(v))$)
            **else if** (*Search*(*Closed*, $v$))
                **if** $(f'(v) < f(v))$
                    *Delete*(*Closed*, $v$); *Insert*(*Open*, $(v, f'(v))$)
            **else** *Insert*(*Open*, $(v, f'(v))$)

Table 15.1: The A* Algorithm.

The algorithm divides the state space in three sets: the set *Open* of visited but not expanded states, the set *Closed* of visited and expanded states, and the set of not already visited states. Similar to Dijkstra's single source shortest path exploration [80], starting with the initial state, A* extracts states from the *Open* set, move them to the *Closed* set and insert their successors in the *Open* set until a goal state is found. In Table 12.1 the differences between Dijkstra's algorithm and A* are underlined. In each expansion step the state with best $f$ value is selected to be expanded next. Nodes that have already been expanded might be encountered on a shorter path. Contrary to Dijkstra's algorithm, A*

deals with the problem by re-inserting the corresponding nodes from the set of already expanded nodes into the *Open* set. This scheme is called *re-opening*.

## 15.3 HSF-SPIN

HSF-SPIN merges the model checker Spin[2] and the heuristic search framework HSF[3]. It is basically an extension of HSF for searching state spaces generated by Promela models.

Like in Spin, two steps must be performed prior to the verification process. The first step generates the source code of the verifier for a given Promela specification. In the second step, the source code is compiled and linked for constructing the verifier. The verifier then checks the model. Among other parameters the user can specify the error type, the search algorithm, and the heuristic estimate as command line options. It is also possible to perform interactive simulations similar to Spin. When verification is done, statistic results are displayed and a solution trail in Spin's format is generated.

HSF-SPIN is based on Spin and its specification language Promela. However, HSF-SPIN is not 100% Promela compatible. Promela specifications with dynamic or non-deterministic process creation are not yet accepted in HSF-SPIN. HSF-SPIN can check all the properties that Spin can validate with the exception of non-progress cycles. HSF-SPIN supports sequential bit-state hashing, but not partial order reduction.

### 15.3.1 A First Example

The HSF-SPIN distribution includes a set of test models. For example, the file `deadlock.philosophers.prm` implements a Promela model of a deadlock solution to Dijkstra's dining philosophers problem. The executable `check` is a verifier of the model, similar to Spin's executable file `pan`. Deadlocks are checked by running the verifier with argument `-Ed` resulting in the following output.

```
HSF-SPIN 1.0
A Simple Promela Verifier based on Heuristic Search Strategies.
This tool is based on Spin 3.4.5  (by G.J. Holzmann) and
                   on HsfLight 2.0 (by S. Edelkamp)
Verifying models/deadlock.philosophers.prm...
Checking for deadlocks with Depth-First Search...
        invalid endstate (at depth 1362)
Printing Statistics...
        State-vector 120 bytes, depth reached 1362, errors: 1
             1341 states, stored
              431 states, matched
             1772 transitions (transitions performed)
               25 atomic steps
             1341 states, expanded
        Range of heuristic was: [0..0]
Writing Trail
Wrote models/deadlock.philosophers.prm.trail
        Length of trail is 1362
```

The verifier runs depth-first search, since it is the default search algorithm. It finds a deadlock at depth 1,362. Following such a long trail is tedious. The A* algorithm (option `-AA`) and a simple heuristic estimate for deadlock detection (option `-Ha`) finds a

[2]`http://netlib.bell-labs.com/netlib/spin/whatispin.html`
[3]`http://www.informatik.uni-freiburg.de/~edelkamp/Hsf`

deadlock at optimal depth 34, expanding and storing less states (17 and 67, respectively), and performing less transitions (73).

### 15.3.2 Compile- and Run-Time Options

The HSF-SPIN verifier accepts only a reduced subset of Spin's compile-time options, for example `-DVECTORSZ` and `-DGCC`. The only specific compile-time option is `-DDEBUG`, to report debug information when running. Each command line argument of HSF-SPIN has the form $-Xx$, where $X$ is the option to be set and $x$ is the value for the option. For example, argument `-Ad` sets the option *search algorithm* to the value *depth-first search*. By giving an option no value, the list of available values for that option is printed. For example, executing `check -A` prints all available search algorithms.

Executing the HSF-SPIN verifier without arguments outputs all available run-time options, e.g. `-Ax`, where $x$ is the search algorithm (A*, IDA*, DFS, NDFS, etc.); `-Ex`, where $x$ is the error to be checked (Deadlock, Assertion, LTL, etc.); and `-Hx`, where $x$ is the heuristic function (Formula-based, Hamming distance, FSM distance, etc.).

## 15.4 Improvement of Trails

Since various explicit on-the-fly model checkers like Spin search the superimposed global state space in depth-first manner, they report the first error that has been encountered even if it appears at a high search depth. One natural option to improve the trail is to impose a shallower depth on the depth-first search engine. However, there are two severe drawbacks to this approach.

The first one is that bounds might increase the search efforts by magnitudes, since a fixed traversal ordering in bounded depth-first exploration in large search depths might miss the lasting error states for a fairly long time. Therefore, even if the first error is found fast, improvements are possibly difficult to obtain. Moreover, to find shorter trails by manual adjusting bounds is time consuming, e.g., trying to improve an optimal witness will fail and result in a full state exploration.

The second drawback, which we call *anomaly in depth-bounded search* (cf. Figure 15.1) is even more crucial to this approach. It can be observed when experimenting with explicit state model checkers that allow the search depth to be limited to a maximum, such as it can be done in SPIN, and in which visited states are kept in a hash-table to avoid an exponential increase in the number of expanded nodes due to the tree expansion of the underlying graph. This implicit pruning results in the fact, that duplicate errors in smaller depths will not necessarily be detected anymore, since they might be blocked by nodes that are already stored. This anomaly emerges frequently in practice when atomic transitions are used, which correspond to potentially long non-branching paths in the search tree. In other words, depth-bounded search with node caching is not complete for error detection in shallower depths than the given bound[4].

We have observed this behavior in some of our models. For example, in a model of a telephony system after establishing a witness of length 756, the search with a new bound

[4]Note that to the contrary, the iterative-deepening variant of A* (IDA*) is complete, since it invokes the depth-first search process starting with the smallest available bound and increasing this bound the smallest possible amount.

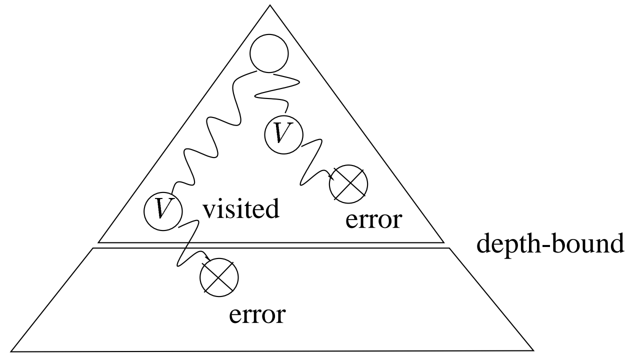


Figure 15.1: Anomaly in Depth-Bounded Search.

755 fails to find one of the remaining error states. For the same Promela model, error detection alternates with different search depths bound: up to bound 67 no error is found, from bound 68 to 139 an error is found, from bound 140 to 154 no error is found, from 155 onwards an error is again found, and so on.

A simple method to correct this anomaly is to enforce *revisiting* of some states. More precisely, a state is revisited (reexplored) when it is reached on a shorter path. Therefore, each state is stored in the hash-table together with its smallest depth value. In fact, this observation was already made for the Spin model checker, in which the anomaly is fixed with the `-DREACH` directive. However, since entire subtree structures for revisited states are re-explored, this method causes a possibly exponential increase in time complexity.

Therefore, we aim at a different aspect of trail improvement; namely heuristic search. The idea is to take the failure state or some of its defining features to set up a heuristic estimate that guides the search process into the direction of that particular state. In contrast to heuristic search strategies described in previous work [107, 109], we exhibit refined information. The main argument is that it is easier to find a specific error situation instead of finding any member according to a general error description. We distinguish two heuristics and two search algorithms. The first heuristic is designed to focus exactly the state that was found in the guidance trail, while the second heuristic relaxes this requirement to important aspects for the given failure type. The two algorithms divide in trail-directed search for safety property violation and trail-directed search for liveness property violation.

### 15.4.1 Hamming Distance Heuristic

Let $S$ be a state of the search space be given in a suitable binary encoding, i.e. as a bit vector $S = (s_1, \ldots, s_k)$. Further on let $S'$ be the desired error state we are searching for. One coarse estimate for the number of transitions necessary to get from $S$ to $S'$ is the number of bit-flips necessary to transform $S$ into $S'$. The estimate is called the *Hamming distance* $H_{HD}(S, S')$, determined by

$$H_{HD}(S, S') = \sum_{i=1}^{k} |s_i - s'_i|$$

Obviously, $|s_i - s'_i| \in \{0, 1\}$ for all $i \in \{1, \ldots, k\}$. Note that the estimate $H_{HD}(S, S')$ is not a lower bound, since one transition might change more than one bit in the state

description at a time. Moreover, the Hamming distance can be refined by taking the binary encoded values of the state variables and their modifiers into account. Nevertheless, the Hamming distance reveals a valuable ordering of the states according to their goal distances.

### 15.4.2 FSM Distance Heuristic

Another distance metric centers around the local states of the finite state machines, which together with the communication queues and variables generate system's global state space.

Let $(pc_1, \ldots, pc_l)$ be the vector of all FSM locations in a state $S$, i.e. $pc_i$, $i \in \{1, \ldots, l\}$, denotes the corresponding program counter. The FSM distance metric $H_{FSM}(S, S')$ according to the goal state $S'$ with FSM state vector $(pc'_1, \ldots, pc'_l)$ is calculated in each FSM separately. When assuming independence of the execution in each finite state machine, we can approximate

$$H_{FSM}(S, S') = \sum_{i=1}^{k} D_i(pc_i, pc'_i)$$

The distances $D_i(pc_i, pc'_i)$ are calculated as the minimal graph theoretical distance from $pc_i$ to $pc'_i$, $i \in \{1, \ldots, l\}$. These values are computed beforehand for each pair of local states with the all-pairs shortest-path algorithm of Floyd-Warshall, so that the retrieval of each value $D_i(pc_i, pc'_i)$ is a constant time operation. In contrast to the Hamming distance, the FSM distance abstracts from the current queue load and from values of the local and global variables. We expect that the search will be then directed into equivalent error states that could potentially be located at smaller search tree depths (see Figure 15.3).

### 15.4.3 Safety Errors

Trail-directed search for safety errors, as visualized in Figure 15.2, takes a trail as an additional input for the model-checker and searches for improvements of its length, especially for a concise and transparent bug-finding process.

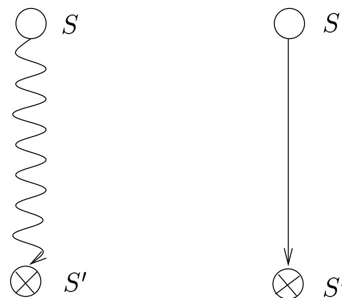


Figure 15.2: Safety Error Trail is Shortened by Trail-Directed Search.

In our case we extract the error state $S'$ to focus the search by the above heuristics $H_{HD}(S, S')$ and $H_{FSM}(S, S')$. These estimates are integrated in the heuristic search algorithm A*. Recall that is complete, and that, if the estimate is a lower bound, the path is optimal.

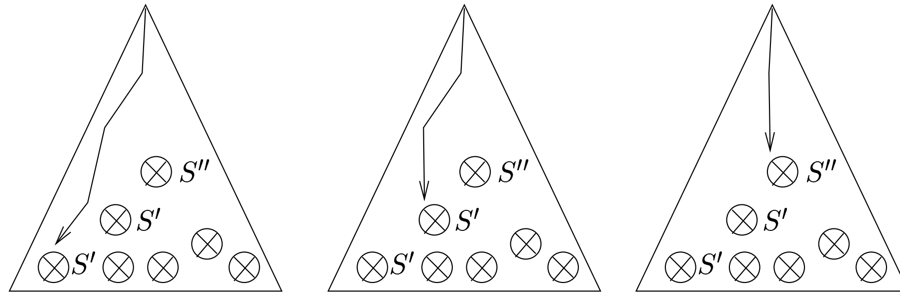


Figure 15.3: Search Trees of Ordinary Search, Full-State Trail-Directed Search, and Partial-State Trail-Directed Search.

Figure 15.3 depicts the search tree inclusive the established trail according to ordinary search, A* with the Hamming distance heuristic $H_{HD}$, and A* with the FSM distance heuristic $H_{FSM}$. Since $H_{HD}$ uses the entire error state description, we call this search *full-state trail-directed search*, while in case of $H_{FSM}$ only a part of the error state description is used, such that this approach is referred to as *partial-state trail-directed search.*

### 15.4.4 Liveness Properties

Remember that a trail to a violated liveness property consists of a path with an initial prefix to a seed state and a cycle starting from that state. Therefore, we can improve the witness trail by trail-directed A*-like search in both parts (cf. Figure 15.4).

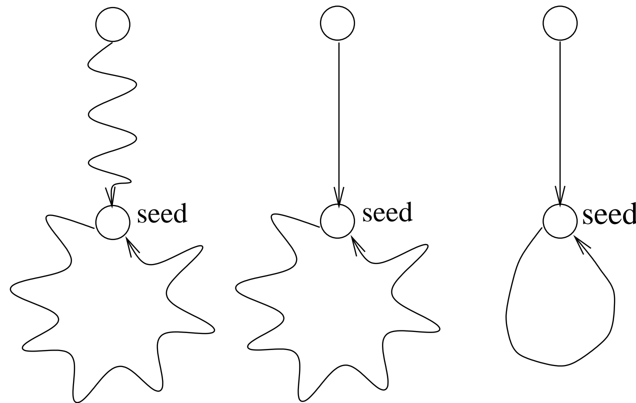


Figure 15.4: Liveness Error Trail Shortened in Two Phases.

In a first improvement phase we search for shortcuts of the path to the seed state. In an independent second phase we perform a cycle-detection search, i.e. a search guided by the seed state from which it has started. In both cases the proposed estimate that we propose is the Hamming distance heuristic $H_{HD}$, since we are searching for the exact seed state, and not for an equivalent one.

## 15.5 Experiments

In our experimental study selected examples for trail improvements are used. We apply the above algorithms to trails obtained by our depth-first search algorithm, producing the same or very similar results to SPIN's depth-first search traversal.

First we consider deadlock detection. As an example we choose the industrial GIOP protocol [201] with a seeded bug and a model of a concurrent program that solves the stable marriage problem [260][5]. Table 15.2 shows that the witness trail is improved to about a half of its original length. The values 67 and 65 in the GIOP model are close to the optimal trail length of 58. In the second case the solution path obtained when using the FSM-based heuristic is near to the optimum of 62 and notably better of the length provided by the algorithm with the Hamming distance heuristic. However, in both examples the search efforts are significantly higher in the case of the FSM-based heuristic than in the case of the Hamming distance heuristic.

| | | DFS | TDA*, $H_{HD}$ | TDA*, $H_{FSM}$ |
|---|---|---|---|---|
| GIOP | Stored States | 326 | 988 | 30,629 |
| (N=2,M=1) | Transitions | 364 | 1,535 | 98,884 |
| | Expanded States | 326 | 432 | 24,485 |
| | Witness Trail | 134 | 67 | 65 |
| Marriers | Stored States | 407,009 | 26,545 | 225,404 |
| (N=4) | Transitions | 1,513,651 | 56,977 | 467,704 |
| | Expanded States | 407,009 | 16,639 | 192,902 |
| | Witness Trail | 121 | 99 | 66 |

Table 15.2: Improving Trails of Deadlocks with Trail-Directed Search in the GIOP and Marriers models.

In the second set of examples we examine another safety property class, namely state invariants. The two protocols we consider are a Promela model of an Elevator system[6] and the POTS telephony protocol model [202]. Table 15.3 shows that the witness trail is shortened by trail-directed search from 510 to 203 and from 756 to 67, respectively. In this case there is no significant difference between the two heuristic estimates.

A *bad sequence* corresponds to a violation of a liveness property. However, it does not reflect a cyclic witness but a simple path. The results in Table 15.4 shows the impact of trail improvement in this scenario for a model of a Fundamental-Mode Circuit (FMC [299]).

The last example is trail improvement for liveness properties that include cycles at seed states in their witness paths. Once more we use the Elevator protocol as a representative example.

Table 15.5 depicts the results of trail-directed search applied to trails obtained by nested depth-first search (NDFS) and the improved version of this algorithm (INDFS). It is shown that cycle seeds are found at smaller depths for the error trails of both algorithms, while the cycle length has not been improved. On the other hand considerable work is

[5]The Promela sources and further information about these models can be obtained from `www.informatik.uni-freiburg.de/~lafuente/models/models.html`

[6]Available from `www.inf.ethz.ch/personal/biere/teaching/mctools/elsim.html`

| | | DFS | TDA*,$H_{HD}$ | TDA*,$H_{FSM}$ |
|---|---|---|---|---|
| Elevator ($N = 3$) | Stored States | 292 | 38,363 | 38,538 |
| | Transitions | 348 | 146,827 | 147,277 |
| | Expanded States | 292 | 38,423 | 38,259 |
| | Witness Trail | 510 | 203 | 203 |
| POTS | Stored States | 506,751 | 2,668 | 2,019 |
| | Transitions | 1,468 $10^6$ | 6,519 | 4,889 |
| | Expanded States | 506,751 | 2,326 | 997 |
| | Witness Trail | 756 | 67 | 67 |

Table 15.3: Improving Trails of Invariants Violation with Trail-Directed Search in the Elevator and POTS models.

| | | DFS | TDA*,$H_{HD}$ | TDA*,$H_{FSM}$ |
|---|---|---|---|---|
| FMC ($N = 3$) | Stored States | 270 | 438 | 419 |
| | Transitions | 364 | 664 | 624 |
| | Expanded States | 279 | 437 | 412 |
| | Witness Trail | 259 | 73 | 73 |

Table 15.4: Improving the Trail of a Bad Sequence in the FMC model.

necessary to improve the length of the trail. Since this is only a single data point more protocols with liveness properties are required for a better judgment.

## 15.6 Conclusions

While previous work on *directed model checking* concentrates on detecting unknown error states, the paradigm of *trail-directed model checking* contemplates the improvement of trails result from error detections, simulations, etc. On the other hand, although paths to errors could be improved with *directed model checking*, the new paradigm proposes richer heuristics based on the information of a singleton given error states. Moreover *directed model checking* is restricted to safety properties, while *trail-directed model checking* is able to improve error trails corresponding to such type of properties.

Trail improvement in our directed model checking tool HSF-SPIN turns out to be an effective aid in software design of concurrent systems. With an acceptable overhead already existing paths are reduced by heuristic search for the established error. The first

| | | NDFS | TDA*,$H_{HD}$ | INDFS | TDA*,$H_{HD}$ |
|---|---|---|---|---|---|
| Elevator ($N = 2$) | Stored States | 171 | 11,205 | 166 | 10,930 |
| | Transitions | 259 | 38,307 | 194 | 37,656 |
| | Expanded States | 208 | 10,901 | 166 | 10,764 |
| | Seed at Depth | 187 | 173 | 177 | 163 |
| | Cycle Length | 90 | 90 | 90 | 90 |
| | Total Length | 277 | 263 | 267 | 253 |

Table 15.5: Improving the Trail of Liveness Property Violation in the Elevator Protocol.

results are promising and put forth the idea of *trail-directed model checking*, that might include more information than the mere description of the error state.

One early approach for focusing trail information is *diagnostic model checking for real-time systems* [233]. It also shifts attention to highlight failure detection, but does not clarify why the established traces are improved compared to ordinary failure trails. Another line of research aims not only to report what went wrong, but explain why it went wrong. However, most approaches in this class such as assumption truth-maintenance systems implemented in the General Diagnostic Engine (GDE [207]) turn out to scale badly.

At the moment we concentrate on SPIN's Promela specification language, but in future we are interested in verifying real software in Java and C. The Bandera tool [69] developed at Kansas University allows slicing of distributed Java-Programs with an export to either SPIN or SMV. The same research line is pursued by the Automated Software Engineering group at NASA Ames Research Center that apply a Java byte code verifier, called Java Path Finder [165]. On the other side, Holzmann [190] has pushed the envelope for actual C-Code verification with the SPIN validator.

# Paper 16

# Partial-Order Reduction in Directed Model Checking

Alberto Lluch-Lafuente, Stefan Edelkamp, and Stefan Leue.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: {edelkamp,lafuente,leue}@informatik.uni-freiburg.de

In *SPIN Workshop*, Lecture Notes in Computer Science. Springer, 2002.

**Abstract**

Partial order reduction is a very succesful technique for avoiding the state explosion problem that is inherent to explicit state model checking of asynchronous concurrent systems. It exploits the commutativity of concurrently executed transitions in interleaved system runs in order to reduce the size of the explored state space. Directed model checking on the other hand addresses the state explosion problem by using guided search techniques during state space exploration. As a consequence, shorter errors trails are found and less search effort is required than when using standard depth-first or breadth-first search. We analyze how to combine directed model checking with partial order reduction methods and give experimental results on how the combination of both techniques performs.

## 16.1 Introduction

Model checking [65] is a formal analysis technique for the verification of hardware and software systems. Given the model of the system as well as a property specification, typically formulated in some temporal logic formalism, the state space of the model is analyzed to check whether the property is valid or not. The main limitation of this method is the size of the resulting state space, known as the *state explosion problem*. It occurs due to non-determinism in the model introduced by data or concurrency.

Different approaches have been proposed to tackle this problem. One of the most successful techniques is partial order reduction [291]. This method explores a reduced state space by exploiting the independence of concurrently executed events. Partial order reduction is particularly efficient in asynchronous systems, where many interleavings of concurrent events are equivalent with respect to a given property specification. Considering only one or a few representatives of one class of equivalent interleavings leads to drastic reductions in the size of the state space to be explored.

Another technique that has been suggested in dealing with the state explosion problem is the use of heuristic search techniques. It applies state evaluation functions to rank the set of successor states in order to decide where to continue the search. Applying such methods often allows to find errors at optimal or sub-optimal depths and to find errors in models for which "blind" search strategies like depth-first and breadth-first search exceed the available time and space resources. Optimal or near-to optimal solutions are particularly important for designers to understand the sequence of steps that lead to an error, since shorter trails are likely to be more comprehensible than longer ones. In protocol verification, heuristic search model checking has been shown to accelerate the search for finding errors [108] and to shorten already existing long trails [110].

It is not a priori obvious to what extent partial order reduction and guided search can co-exist in model checking. In fact, as we show later, applying partial-order reduction to a state space does not preserve optimiality of the shortest path to a target state. It is the goal of this paper to show that nevertheless, partial order reduction and directed model checking can co-exist, and that the mutual negative influence is only minimal.

In this paper, we will focus on safety error detection in model checking. We will establish a hierarchy of relaxation of the cycle condition for partial order reduction known as C3, and we will classify the relaxations with respect to their applicability to different classes of heuristic search algorithms. To the best of our knowledge, at the time of writing no publication addressing heuristic search in model checking [108, 109, 110, 186, 68, 239, 352] has analyzed how to combine guided search with partial order reduction.

The paper is structured as follows. Section 16.2 gives some background on directed model checking. Section 16.3 discusses partial order reduction and a hierarchy of conditions for its application to different search algorithms. This Section also addresses the problem of optimality in the length of the counterexamples. Section 16.4 presents experimental results showing how the combination of partial order reduction and directed model checking perform. Section 16.5 summarizes the results and concludes the paper.

## 16.2 Directed Model Checking

Analysts have different expectations regarding the capabilities of formal analysis tools at different stages of the software process [68]. In *exploratory mode*, usually applicable to earlier stages of the process, one wishes to find errors fast. In *fault-finding mode*, which usually follows later, one expects to obtain meaningful error trails while one is willing to tolerate somewhat longer execution times. Directed model checking has been identified as an improvement of standard model checking algorithms that help in achieving the objectives of both modes.

Early approaches to directed model checking [239, 352] propose the use of best-first search algorithms in order to accelerate the search for error states. Further approaches [109, 108, 110, 68] propose the full spectrum of classical heuristic search strategies for the verification process in order to accelerate error detection and to provide optimal or near-to-optimal trails. Most of these techniques can be applied to the detection of safety properties only or for shortening given error traces corresponding to liveness violations [110].

Contrary to blind search algorithms like depth- and breadth-first search, heuristic search exploits information of the specific problem being solved in order to guide the search. Estimator functions approximate the distance from a given state to a set of goal states. The values provided by these functions decide in which direction the search will be continued. Two of the most frequently used heuristic search algorithms are A* [161] and IDA* [213]. In the following we describe a general state expanding search algorithm that can be either instantiated as a depth or breadth first search algorithm or as one of the above heuristic search algorithms. We briefly the basic principles of heuristic search algorithms, and consider different heuristic estimates to be applied in the context of directed model checking for error detection. In our setting we interpret error states as goal nodes in an underlying graph representation of the state space with error trails corresponding to solution paths.

### 16.2.1 General State Expanding Search Algorithm

The general state expanding search algorithm of Figure 16.1 divides the state space $S$ into three sets: the set $Open$ of visited but not yet expanded states, the set $Closed$ of visited and expanded states and the set $S \setminus (Closed \cup Open)$ of not yet visited states. The algorithm performs the search by extracting states from $Open$ and moving them into $Closed$. States extracted from $Open$ are expanded, i.e. their successor states are generated. If a successor of an expanded state is neither in $Open$ nor in $Closed$ it is added to $Open$.

Breadth-first search and depth-first search can be defined as concrete cases of the general algorithm presented above, where the former implements $Open$ as a FIFO queue and the latter as a stack.

### 16.2.2 Algorithm A*

Algorithm A* treats $Open$ as a priority queue in which the priority of a state $v$ is given by function $f(v)$ that is computed as the sum of the length of the path from the start state $g(v)$ and the estimated distance to a goal state $h(v)$. In addition to the general algorithm, A* can move states from $Closed$ to $Open$ when they are reached along a

```
procedure search
  Closed ← ∅
  Open ← ∅
  Open.insert(start_state)
  while (Open ≠ ∅) do
    u ← Open.extract()
    Closed.insert(u)
    if error(u) then
      exit ErrorFound
    for each successor v of u do
      if not v ∈ Closed ∪ Open then
          Open.insert(v)
```

Figure 16.1: A general state expanding search algorithm.

shorter path. This step is called reopening and is necessary to guarantee that the algorithm will find the shortest path to the goal state when non-monotone heuristics are used. For the sake of simplicity, throughout the paper we only consider monotone heuristics which do not require reopening [287]. Monotone heuristics satisfy that for each state $u$ and each successor $v$ of $u$ the difference between $h(u)$ and $h(v)$ is less or equal to the cost of the transition that goes from $u$ to $v$. Assuming monotone estimates is not a severe restriction, since it turns out that in practical examples most proposed heuristics are indeed monotone. If $h$ is a lower bound of the distance to a goal state, then A* is admissible, which means that it will always return the shortest path to a goal state [278]. Best-first search is a common variant of A* that takes only $h$ into account.

### 16.2.3 Iterative-deepening A*

*Iterative-deepening A**, IDA* for short, is a refinement of the brute-force depth-first iterative deepening search (DFID) that combines the space efficiency of depth-first search and the admissibility of A*. While DFID performs successive depth-first search iterations with increasing depth bound, in IDA* increasing cost bounds are used to limit search iterations. The cost bound $f$ of a state is the same as in A*. Similar to A*, IDA* guarantees optimal solution paths if the estimator is a lower bound.

### 16.2.4 Heuristic Estimates

The above presented search algorithms require suitable estimator functions. In model checking, such functions approximate the number of transitions for the system to reach an error state from a given state. During the model checking process, however, an explicit error state is not always available. In fact, in many cases we do not know if there is an error in the model at all. We distinguish the cases when errors are unknown and when error states are explicit.

If an explicit error state is given, estimates that exploit the information of this state can be devised. Examples are estimates based on the Hamming distance of the state vectors for the current and the targe state, and the FSM distance that uses the minimum local distances in the state transition graph of the different processes to derive an estimat [110].

Estimating the distance to *unknown* error states is more difficult. The formula-based heuristic [108] constructs a function that characterizes error states. Given an error formula $f$ and starting from state $s$, a heuristic function $h_f(s)$ is constructed for estimating the number of transitions needed until a state $s'$ is reached, where $f(s')$ holds. Constructing an error formula for deadlocks is not trivial. In [108] we discuss various alternatives, including formula based approaches, an estimate based on the number of non-blocked processes, and an estimate derived from user-provided characterizations of local control states as deadlock-prone.

# 16.3 Partial Order Reduction

Partial order reduction methods exploit the commutativity of asynchronous systems in order to reduce the size of the state space. The resulting state space is constructed in such a manner that it is equivalent to the original one with respect to the specification. Several partial order approaches have been proposed, namely those based on "stubborn" sets [342], "persistent" sets [146] and "ample" sets [290]. Although they differ in detail, they are based on similar ideas. Due to its popularity, in this paper we mainly follow the ample set approach. Nonetheless, most of the reasoning presented in this paper can be adjusted to any of the other approaches.

## 16.3.1 Stuttering Equivalence of Labeled Transition Systems

Our approach is mainly focused to the verification of asynchronous systems where the global system is constructed as the asynchronous product of a set of local component processes following the interleaving model of execution. Such systems can be modeled by labeled transitions systems.

A labeled finite transition system is a tuple $\langle S, S_0, T, AP, L\rangle$ where $S$ is a finite set of states, $S_0$ is the set of initial states, $T$ is a finite set of transitions such that each transition $\alpha \in T$ is a partial function $\alpha : S \to S$, $AP$ is a finite set of propositions and $L$ is a labeling function $S \to 2^{AP}$. The execution of a transition system is defined as a sequence of states interleaved by transitions, i.e. a sequence $s_0\alpha_0 s_1\ldots$, such that $s_0$ is in $S_0$ and for each $i \geq 0$, $s_{i+1} = \alpha_i(s_i)$.

The algorithm for generating a reduced state space explores only some of the successors of a state. A transition $\alpha$ is *enabled* in a state $s$ if $\alpha(s)$ is defined. The set of enabled transitions from a state $s$ is usually called the *enabled set* and denoted as $enabled(s)$. The algorithm selects and follows only a subset of this set called the *ample set* and denoted as $ample(s)$. A state $s$ is said to be *fully expanded* when $ample(s) = enabled(s)$.

Partial order reduction techniques are based on the observation that the order in which some transitions are executed is not relevant. This leads to the concept of independence between transitions. Two transitions $\alpha, \beta \in T$ are independent if for each state $s \in S$ in which both transitions are defined the following two properties hold:

1. $\alpha$ and $\beta$ do not disable each other: $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$.

2. $\alpha$ and $\beta$ are *conmutative*: $\alpha(\beta(s)) = \beta(\alpha(s))$.

Two transitions are dependent if they are not independent. A further fundamental concept is the fact that some transitions are *invisible* with respect to atomic propositions that occur in the property specification. A transition $\alpha$ is invisible with respect to a set of propositions $P$ if for each state $s$ and for each successor $s'$ of $s$ we have $L(s) \cap P = L(s') \cap P$ .

We now present the concept of *stuttering equivalence* with respect to a property P. A *block* is defined as a finite execution containing invisible transitions only. Intuitively, two executions are stuttering equivalent if they can be defined as a concatenation of blocks such that the atomic propositions of the $i$-th block of boths executions have the same intersection with $P$, for each $i > 0$. Figure 16.2 depicts two stuttering equivalent paths with respect to a property in which only propositions $p$ and $q$ occurr.

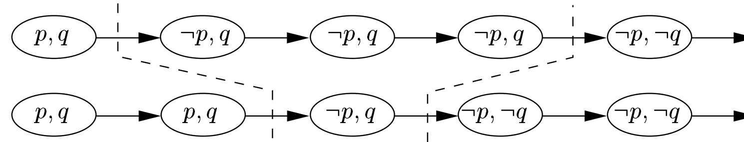

Figure 16.2: Stuttering equivalent executions.

Two transition systems are stuttering equivalent if and only if they have the same set of initial states and for each execution in one of the systems starting from an initial state there exists a stuttering equivalent execution in the other system starting from the same initial state. It can be shown that LTL$_{-X}$ formulae[1] cannot distinguish between stuttering equivalent transition systems [65]. In other words, if $M$ and $N$ are two stuttering equivalent transition systems, then $M$ satisfies a given LTL$_{-X}$ specification if and only if $N$ also does.

### 16.3.2 Ample Set Construction for Safety LTL$_{-X}$

The main goal of the ample set construction is to select a subset of the successors of every state such that the reduced state space is stuttering equivalent to the full state space with respect to a property specification given by a set $P$ of atomic propositions. The construction should offer significant reduction without requiring a high computational overhead. The following four conditions are necessary and sufficient for the proper construction of a partial order reduced state space for a given property specification $P$.

**Condition C0**: $ample(s)$ is empty exactly when $enabled(s)$ is empty.

**Condition C1**: Along every path in the full state space that starts at $s$, a transition that is dependent on a transition in $ample(s)$ does not occur without a transition in $ample(s)$ occurring first.

**Condition C2**: If a state $s$ is not fully expanded, then each transition $\alpha$ in the ample set must be invisible with regard to $P$.

[1]LTL$_{-X}$ is the linear time temporal logic without the next-time operator $X$.

**Condition C3**: If for each state of a cycle in the reduced state space, a transition $\alpha$ is enabled, then $\alpha$ must be in the ample set of some of the states of the reduced state space.

Observe that the approximations used in **C0**, **C1** and **C2** can be implemented independently from the particular search algorithm used. It was shown in [65] that the complexity of checking **C0** and **C2** does not depend on the search algorithm. Checking Condition **C1** is more complicated. In fact, it has been shown to be at least as hard as checking reachability for the full state space. It is, however, usually over-approximated by checking a stronger condition [65] that can be checked independently of the search algorithm.

Condition **C3** has been implicitly proposed in [185]. In the following we focus on this condition. We will see that the complexity of checking it depends on the search algorithm used.

### 16.3.3 Dynamically Checking C3

Checking **C3** can be reduced to detecting cycles during the search. Cycles can easily be established in depth-first search: Every cycle contains a *backward edge*, i.e. an edge that links back to a state that is stored on the search stack [65]. Consequently, avoiding ample sets containing only backward edges except when the state is fully expanded ensures satisfaction of **C3** when using depth-first search or IDA*, since both methods perform a depth-first traversal. The resulting stack-based characterization **C3**$_{stack}$ can be stated as follows[185]:

**Condition C3**$_{stack}$: If a state $s$ is not fully expanded, then at least one transition in $ample(s)$ does not lead to a state on the search stack.

The implementation of **C3**$_{stack}$ for depth-first search strategies marks each expanded state on the stack with an additional flag, so that stack containment can be checked in constant time. Depth-first strategies that record visited states will not consider every cycle in the state space on the search stack, since there might exist exponentially many of them. However, **C3**$_{stack}$ is still a sufficient condition for **C3** since every cycle contains at least a backward edge.

Detecting cycles with general state expanding search algorithms that do not perform a depth-first traversal of the state space is more complex. For a cycle to exist, it is necessary to reach an already visited state. If during the search a state is found to be already visited, checking that this state is part of a cycle requires to check if this state is reachable from itself which increases the time complexity of the algorithm from linear to quadratic in the size of the state space. Therefore the commonly adopted approach assumes that a cycle exists whenever an already visited state is found. Using this idea leads to weaker reductions, since it is known that the state spaces of concurrent systems usually have a high density of duplicate states. The resulting condition [58, 10] is defined as:

**Condition C3**$_{duplicate}$: If a state $s$ is not fully expanded, then at least one transition in $ample(s)$ does not lead to an already visited state.

A proof of sufficiency of condition **C3**$_{stack}$ for depth-first search is given in [185]. The proof of sufficiency of condition **C3**$_{duplicate}$ when combined with a depth-first search is

given by the fact that $\mathbf{C3}_{duplicate}$ implies $\mathbf{C3}_{stack}$; if at least one transition in $ample(s)$ has a non-visited successor this transition certainly does not lead to a successor on the stack.

In order to prove the correctness of partial order reduction with condition $\mathbf{C3}_{duplicate}$ for general state expanding algorithms in the following lemma, we will use induction on the state expansion ordering, starting from a completed exploration and moving backwards with respect to the traversal algorithm. As a by-product the more general setting in the lemma also proves the correctness of partial order reduction according to condition $\mathbf{C3}_{duplicate}$ for depth-first, breadth-first, best-first, and A* like search schemes. The lemma fixes a state $s \in S$ after termination of the search and ensures that each enabled transition is executed either in the ample set or in a state that appears later on in the expansion process. Therefore, no transition is omitted. Applying the lemma to all states $s$ in $S$ implies **C3**, which, in turn, ensures a correct reduction.

**Lemma 13** *For each state $s \in S$ the following is true: when the search of a general search algorithm terminates, each transition $\alpha \in enabled(s)$ has been selected either in $ample(s)$ or in a state $s'$ such that $s'$ has been expanded after $s$.*

**Proof:** Let $s$ be the last expanded state. Every transition $\alpha \in enabled(s)$ leads to an already expanded state, otherwise the search would have been continued. Condition $\mathbf{C3}_{duplicate}$ enforces therefore that state $s$ is fully expanded and the lemma trivially holds for $s$.

Now suppose that the lemma is valid for those states whose expansion order is greater than $n$. Let $s$ be the $n$-th expanded state. If $s$ is fully expanded, the lemma trivially holds for $s$. Otherwise we have that $ample(s) \subset enabled(s)$. Transitions in $ample(s)$ are selected in $s$. Since $ample(s)$ is accepted by condition $\mathbf{C3}_{duplicate}$ there is a transition $\alpha \in ample(s)$ such that $\alpha(s)$ leads to a state that has not been previously visited nor expanded. Evidently the expansion order of $\alpha(s)$ is higher than $n$. Condition **C1** implies that the transitions in $ample(s)$ are all independent from those in $enabled(s) \setminus ample(s)$ [65]. A transition $\gamma \in enabled(s) \setminus ample(s)$ cannot be dependent from a transition in $ample(s)$, since otherwise in the full graph there would be a path starting with $\gamma$ and a transition depending on some transition in $ample(s)$ would be executed before a transition in $ample(s)$. Hence, transitions in $enabled(s) \setminus ample(s)$ are still enabled in $\alpha(s)$ and contained in $enabled(\alpha(s))$. By the induction hypothesis the lemma holds for $\alpha(s)$ and, therefore, transitions in $enabled(s) \setminus ample(s)$ are selected in $\alpha(s)$ or in a state that is expanded after it. Hence the lemma also holds for $s$. □ ■

## 16.3.4 Statically Checking C3

In contrast to the previous approaches the method to be presented in this Section explicitly exploits the structure of the underlying interleaving system. Recall that the global system is constructed as the asynchronous composition of several components. The authors of [228] present what they call a *static* partial order reduction method based on the following observation. Any cycle in the global state space is composed of a local cycle, which may be of length zero, in the state transition graph of each component process. Breaking every local cycle breaks every global cycle. The structure of the processes of the system is analyzed before the global state space generation begins. The method is independent from the search algorithm to be used during the verification.

Some transitions are marked are with a special flag, called *sticky*. Sticky transitions enforce full expansion of a state. Marking at least one transition in each local cycle as *sticky* guarantees that at least one state in each global cycle is fully expanded, which satisfies condition **C3**. The resulting **C3**$_{static}$ condition is defined as follows:

**Condition C3**$_{static}$: If a state $s$ is not fully expanded then no transition in $ample(s)$ is sticky.

Selecting one sticky transition for each local cycle is a naive approach that can be made more effective. The impact of local cycles on the set of variables of the system can be analyzed in order to establish certain dependencies between local cycles. For example, if a local cycle $C_1$ has an overall incrementing effect on a variable $v$, for a global cycle to exist, it is necessary (but not sufficient) to execute $C_1$ in combination with a local cycle $C_2$ that has an overall decrementing effect on $v$. In this case one can select only a sticky transition for this pair of local cycles.

### 16.3.5 Hierarchy of C3 Conditions

Figure 16.3 depicts a diagram with all the presented **C3** conditions. Arrows indicate which condition enforeces which other. In the rest of the paper we will say that a condition $A$ is stronger than a condition $B$ if $A$ enforces $B$. The dashed arrow arrow from $C3_{duplicate}$ to **C3**$_{stack}$ denotes that when search is done with a depth-first search based algorithm **C3**$_{stack}$ enforces $C3_{duplicate}$ but not viceversa. The dashed regions contain the conditions that can be correctly used with general state expanding algorithms, and those that work only for depth-first search like algorithms. For a given algorithm, the arrows also denote that a condition will produce better or equal reduction.

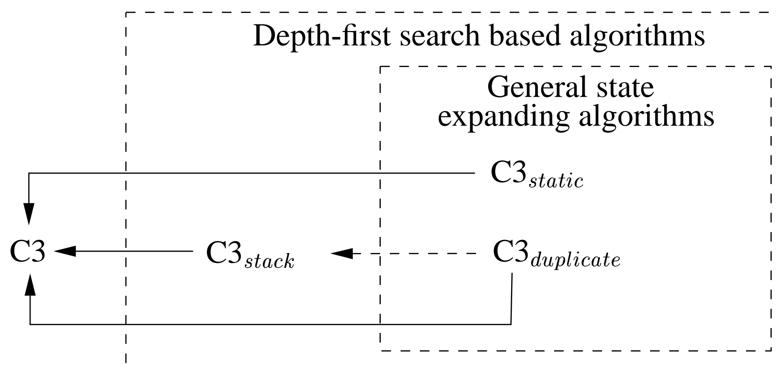


Figure 16.3: C3 conditions.

### 16.3.6 Solution Quality and Partial Order

One of the goals of directed model checking is to find shortest paths to errors. Although from a practical point of view near-to optimal solutions may be sufficient to help designers during the debugging phase, finding optimal counterexamples still remains an important theoretical question. Heuristic search algorithms require lower bound estimates for guaranteeing optimal solution lengths.

Partial order reduction, however, does not preserve optimality of the solution length for the full state space. In fact, the shortest path to an error in the reduced state space may be longer than the shortest path to an error state in the full state space. Intuitively, the reason is that the concept of stuttering equivalence does not make assumptions about the length of the equivalent blocks. Suppose that the transitions $\alpha$ and $\beta$ of the state space depicted in Figure 16.4 are independent and that $\alpha$ is invisible with respect to the set of propositions $p$. Suppose further that $p$ is the only atomic proposition occurring in the safety property we want to check. With these assumptions the reduced state space for the example is stuttering equivalent to the full one. The shortest path that violates the invariant in the reduced state space is $\alpha\beta$, which has a length of 2. In the full one the path $\beta$ is the shortest path to an error state and the error trail has a length of 1. Section 16.4 presents experimental evidence for a reduction in solution quality when applying partial order reduction.

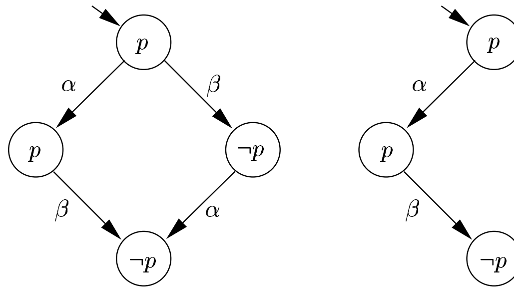


Figure 16.4: Example of a full state space (left) and a reduction (right).

## 16.4 Experiments

The experimental results that we report in this Section have been obtained using our experimental directed model checker HSF-SPIN[2] [108] for which we have implemented the described reduction methods. All results were produced on a SUN workstation, UltraSPARC-II CPU with 248 Mhz.

We use a set of Promela models as benchmarks including a model of a leader election protocol[3] [84] (leader), the CORBA GIOP protocol [201] (giop), a telephony model[4] [202] (pots), and a model of a concurrent program that solves the stable marriage problem [260] (marriers). The considered versions of these protocols violate certain safety properties.

### 16.4.1 Exhaustive Exploration

The objective of the first set of experiments is to show how the different variants of the **C3** condition perform. We expect that stronger **C3** conditions according to hierarchy in Figure 16.3 lead to weaker reductions in the number of stored and expanded states and transitions.

[2]Available at `www.informatik.uni-freiburg.de/ lafuente/hsf-spin`

[3]Available at `netlib.bell-labs.com/netlib/spin`

[4]Available at `www.informatik.uni-freiburg.de/ lafuente/models/models.html`

| Model | Reduction | States | Transitions | Time |
|---|---|---|---|---|
| marriers | No Reduction | 96,295 | 264,053 | 20.6 |
| | $\mathbf{C3}_{stack}$ | 29,501 | 37,341 | 5.5 |
| | $\mathbf{C3}_{duplicate}$ | 72,536 | 111,170 | 17.5 |
| | $\mathbf{C3}_{static}$ | 57,067 | 88,119 | 10.7 |
| leader | No Reduction | 54,216 | 210,548 | 36.3 |
| | $\mathbf{C3}_{stack}$ | 963 | 4,939 | 4.4 |
| | $\mathbf{C3}_{duplicate}$ | 1,417 | 6,899 | 5.0 |
| | $\mathbf{C3}_{static}$ | 2,985 | 7,527 | 4.8 |
| giop | No Reduction | 664,376 | 2,579,722 | 259.3 |
| | $\mathbf{C3}_{stack}$ | 65,964 | 90,870 | 23.1 |
| | $\mathbf{C3}_{duplicate}$ | 284,083 | 605,147 | 115.0 |
| | $\mathbf{C3}_{static}$ | 231,102 | 445,672 | 79.0 |

Table 16.1: Exhaustive exploration with depth-first search and several reduction methods.

We use the marriers, leader and giop protocols in our experiments. The pots model is too large to be explored exhaustively. In this and all following experiments we have selected the biggest configuration of these protocols that can still be exhaustively analyzed. Exploration is performed by depth-first search.

Table 16.1 depicts the size of the state space as a result of the application of different **C3** conditions. The number of transitions performed and the running time in seconds are also included. For each model, the first row indicates the size of the explored state space when no reduction is used.

As expected stronger conditions offer weaker reductions. This loss of reduction is especially evident in the giop protocol, where the two conditions potentially applicable in A*, namely $\mathbf{C3}_{duplicate}$ and $\mathbf{C3}_{static}$, are worse by about a factor of 4 with respect to the condition that offers the best reduction, namely $\mathbf{C3}_{stack}$.

For the marriers and giop protocols the static reduction yields a stronger reduction than condition $\mathbf{C3}_{duplicate}$. Only for the leader election algorithm this is not true. This is probably due to the relative high number of local cycles in the state transition graph of the processes in this model, and to the fact that there is no global cycle in the global state space. Since our implementation of the static reduction considers only the simplest approach where one transition in each cycle is marked as sticky, we assume that the results will be even better with refined methods for characterizing transitions as sticky.

In addition to the reduction in space consumption, partial order reduction also provides reduction in time. Even though the overhead introduced by the computation of the ample set and the static computation prior to the exploration when $\mathbf{C3}_{static}$ is used, time reduction is still achieved in all cases.

### 16.4.2 Error Finding with A* and Partial Order Reduction

The next set of experiments is intended to highlight the impact of various reduction methods when detecting errors with A*. More precisely, we want to compare the two **C3** conditions $\mathbf{C3}_{duplicate}$ and $\mathbf{C3}_{static}$ that can be applied jointly with A*.

Table 16.2 shows the effect of applying $\mathbf{C3}_{duplicate}$ and $\mathbf{C3}_{static}$ in conjunction with A*. The table has the same format as the previous one, but this time the length of the error

| Model | Reduction | States | Transitions | Time | Length |
|---|---|---|---|---|---|
| marriers | no | 5,077 | 12,455 | 0.93 | 50 |
| | $\mathbf{C3}_{duplicate}$ | 2,988 | 4,277 | 0.51 | 50 |
| | $\mathbf{C3}_{static}$ | 1,604 | 1,860 | 0.31 | 50 |
| pots | no | 2,668 | 6,519 | 1.57 | 67 |
| | $\mathbf{C3}_{duplicate}$ | 1,662 | 3,451 | 1.08 | 67 |
| | $\mathbf{C3}_{static}$ | 1,662 | 3,451 | 1.00 | 67 |
| leader | no | 7,172 | 22,876 | 6.87 | 58 |
| | $\mathbf{C3}_{duplicate}$ | 65 | 3,190 | 4.76 | 77 |
| | $\mathbf{C3}_{static}$ | 399 | 3,593 | 4.88 | 66 |
| giop | no | 31,066 | 108,971 | 26.50 | 58 |
| | $\mathbf{C3}_{duplicate}$ | 21,111 | 48,870 | 16.68 | 58 |
| | $\mathbf{C3}_{static}$ | 12,361 | 24,493 | 9.36 | 58 |

Table 16.2: Finding a safety violation with A* and several reduction methods.

trail is included. Similar to SPIN[5], we count a sequence of atomic steps (respectively expansions) as one unique transition (expansion), but length of the error trail is given in steps in order to provide a better idea of what the user of the model checker gets.

As expected, both conditions achieve a reduction in the number of stored states and transitions performed. Solution quality is only lost in the case of leader. This occurs also in experiments done with IDA*. In the same test case $\mathbf{C3}_{static}$ requires the storage of more states and the execution of more transitions than $\mathbf{C3}_{duplicate}$. The reasons are the same as the ones mentioned in the previous set of experiments. On the other hand, $\mathbf{C3}_{duplicate}$ produces a longer error trail. A possible interpretation is that more reduction leads to higher probability that the anomaly that causes the loss of solution quality occurs. In other words, the bigger the reduction is, the longer the stuttering equivalent executions and, therefore, the longer the expected trail lengths become. Table 16.2 also shows that the overhead introduced by partial order reduction and heuristic search does not avoid time reduction.

### 16.4.3 Error Finding with IDA* and Partial Order Reduction

We also investigated the effect of partial order reduction when the state space exploration is performed with IDA*. The test cases are the same of the previous Section. Partial order reduction with IDA* uses the cycle condition $\mathbf{C3}_{stack}$.

Table 16.3 depicts the results on detecting a safety error with and without applying partial order reduction. The table shows the total number of transitions performed, the maximal peak of stored states and the length of the provided counterexamples. As in the previous set of experiments, solution quality is only lost when applying partial order reduction in the leader election algorithm. On the other hand, this is also the protocol for which the best reduction is obtained. We assume that the reason is the same as indicated in the previous set of experiments. In addition, the overhead introduced by partial order reduction and heuristic search does avoid time reduction as explained for previous experiments.

[5] Available at `netlib.bell-labs.com/netlib/spin/whatispin.html`

| Model | Reduction | States | Transitions | Time | Length |
|---|---|---|---|---|---|
| marriers | no | 4,724 | 84,594 | 19.29 | 50 |
| | yes | 1,298 | 4,924 | 8.40 | 50 |
| pots | no | 2,422 | 46,929 | 36.52 | 67 |
| | yes | 1,518 | 20,406 | 28.37 | 67 |
| leader | no | 6,989 | 141,668 | 210.67 | 56 |
| | yes | 55 | 50,403 | 73.90 | 77 |
| giop | no | 30,157 | 868,184 | 225.54 | 58 |
| | yes | 7,441 | 102,079 | 78.43 | 58 |

Table 16.3: Finding a safety violation with IDA* with and without reduction.

### 16.4.4 Combined Effect of Heuristic Search and Partial Order

In this Section we are interested in analyzing the combined state space reduction effect of partial order reduction and heuristic search. More precisely, we have measured the reduction ratio (size of full state space vs. size of reduced state space) provided by one of the techniques when the other technique is enabled or not, as well as the reduction ratio of using both techniques simultaneously.

| marriers | N | C |
|---|---|---|
| H | 2.3 | 6.5 |
| PO | 40.8 | 117.6 |
| H+PO | 267.0 | |
| **pots** | **N** | **C** |
| H | 5.9 | 8.4 |
| PO | 1.4 | 1.6 |
| H+PO | 9.5 | |
| **leader** | **N** | **C** |
| H | 1.9 | 2.6 |
| PO | 2.7 | 3.2 |
| H+PO | 5.9 | |
| **giop** | **N** | **C** |
| H | 1.3 | 1.3 |
| PO | 2.6 | 2.5 |
| H+PO | 3.3 | |

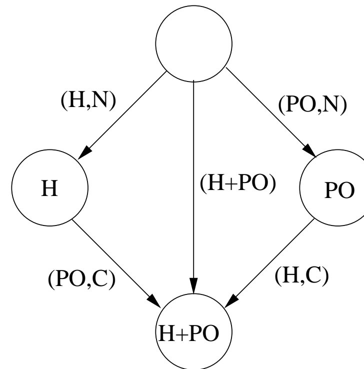


Figure 16.5: Table with reduction factor due partial order and heuristic search (left) and an explanatory diagram (right).

The Table on the left of Figure 16.5 indicates the reduction factor achieved by partial order and heuristic search when A* is used as the search algorithm. The Figure also includes a diagram that helps to understand the table. The reduction factor due to a given technique is computed as the number of stored states when the search is done without applying the respective technique divided by the number of stored states when the search is done applying the technique. Recall that when no heuristic is applied, A* performs breadth-first search. A search is represented in the diagram by a circle labeled with the applied technique(s), namely heuristic search (H), partial-order reduction (PO) or both (H+PO). The labels of the edges in the diagram refer to the cells of the table which

contain the measured reduction factors. The leftmost column of the table indicates the technique(s) for which the reduction effect is measured. When testing the reduction ratios of the methods separately, we distinguish whether the other method is applied (C) or not (N).

In some cases the reduction factor provided by one of the techniques when working alone ((H,N) and (PO,N)) improves when the other technique is applied ((H,C) and (PO,C)). This is particularly evident in the case of the marriers model, where the reduction provided by heuristic search is improved from 2.3 to 6.5 and that of partial order reduction increases from 40.8 to 117.6. The expected gain of applying both independently would be $2.3 \times 40.8 = 93.8$ while the combined effect is a reduction of $267.0$ which indicates a synergetic effect. However, as illustrated by the figures for the giop model, synergetic gains cannot always be expected.

## 16.5 Conclusions

When combining partial order reduction with directed search two main problems must be considered. First, common partial order reduction techniques require to check a condition which entails the detection of cycles during the construction of the reduced state space. Depth-first search based algorithms like IDA* can easily detect cycles during the exploration. On the other side, heuristic search algorithms like A* are not well-suited for cycle detection. Stronger cycle conditions or static reduction methods have to be used. We have established a hierarchy of approximation conditions for ample set condition **C3** and our experiments show that weaker the condition, the better the effect on the state space search.

Second, partial order reduction techniques do not preserve optimality of the length of the path to error states. In other words, when partial order is used there is no guarantee to find the shortest counterexample that lead to an error, which is one of the core objectives of the paradigm of directed model checking. In current work we are analyze the possibility of avoiding this problem by exploiting independence of events to shorten error trails.

Experimental results show that in some instances, partial order reduction has positive effects when used in combination with directed search strategies. Although solution quality is lost in some cases, significant reductions can be achieved even when using A* with weaker methods than classical cycle conditions. Static reduction, in particular, seems to be more promising than other methods applicable with A*. Partial order reduction provides drastic reductions when error detection is performed by IDA*. We have also analyzed the combined effect of heuristics and reduction, showing than in most cases the reduction effect of one technique is lightly accentuated by the other. Experimental results also show that both techniques reduce running time even though the overhead they introduce.

# Part VI

# Hardware Verification

# Paper 17

# Error Detection with Directed Symbolic Model Checking

Frank Reffel
Institut für Logik, Komplexität und Deduktionssysteme,
Universität Karlsruhe,
Am Fasanengarten 5, D-76128 Karlsruhe
eMail: reffel@ira.uka.de

Stefan Edelkamp.
Institut für Informatik,
Universität Freiburg,
Georges-Köhler-Allee 51,
D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

In *World Congress on Formal Methods (FM)*, Lecture Notes in Computer Science, pages 195–211. Springer, 1999.

**Abstract**

In practice due to entailed memory limitations the most important problem in model checking is state space explosion. Therefore, to prove the correctness of a given design binary decision diagrams (*BDDs*) are widely used as a concise and symbolic state space representation. Nevertheless, *BDDs* are not able to avoid an exponential blow-up in general. If we restrict ourselves to find an error of a design which violates a safety property, in many cases a complete state space exploration is not necessary and the introduction of a heuristic to guide the search can help to keep both the explored part and the associated *BDD* representation smaller than with the classical approach.

In this paper we will show that this idea can be extended with an automatically generated heuristic and that it is applicable to a large class of designs. Since the proposed algorithm can be expressed in terms of *BDDs* it is even possible to use an existent model checker without any internal changes.

## 17.1 Introduction

To formulate the specification properties of a given design many different temporal logics are available, each of them with a different expressive power: (Fair-) CTL [64] is a branching time logic, LTL [237] is a linear time logic and CTL* [118] is a superset containing both of them. CTL* itself is a subset of the $\mu$-calculus [224] which in addition allows to verify bisimulation and other more complex properties.

In practice, however, the characteristics people mainly try to verify are simple safety properties that are expressible in all of the logics mentioned above. They can be checked through a simple calculation of all reachable states. Unfortunately, this computation can become intractable for systems consisting of several asynchronously interacting modules.

Although *BDDs* [50] allow a succinct representation of a system they cannot always avoid an increase in *BDD*-sizes caused by the typical exponential blow-up of states. However, model checking is not only used to show the correctness of a complete system, but also as a very efficient method to find errors during the construction phase in order to avoid cost intensive correction phases later on.

In early design phases a system typically contains many errors such that nobody would expect a successful verification. We should try to detect these errors as soon as possible to avoid the calculation of the entire state space. Local model checking methods [125] attempt to exploit only a small part of the state space while global model checking techniques usually calculate all reachable states. Moreover, their fix-point calculation requires a backward traversal and a lot of work is spent in treating unreachable states. Hence, in order just to detect an error local model checking methods [333] can be more efficient. So a suitable application of model checking can replace parts of the classical debugging and testing work because it allows the detection of more errors in less time.

The method proposed in this paper focuses on safety properties. Starting with the set of initial states it performs a forward traversal of the system and exploits only that part of the set of reachable states that is most likely to lead to an error state. This is sufficient to construct a counter example of the violated property helping the designer to understand and fix the failure of the system. To guide the search a heuristic estimates the number of transition steps necessary to reach the error state. If the heuristic fulfills a certain property it guarantees the detection of a minimal counter example.

Our algorithm detects errors in systems unable to be verified by traditional symbolic model checking since the *BDDs* exhaust the available memory resources. Even if we assume pure forward traversal, after several iterations not containing an error state the large amount of states that has to be stored by an unguided search becomes too big; while heuristic search finds the error within an acceptable amount of time without suffering from memory problems.

Since all states have to be visited, our method fails to entirely validate a correct system, but this should be postponed until the end of the construction phase when most of the errors have been removed and the correctness of the system is more probable. The successful verification of large systems can be a very time consuming work which requires elaborated methods and a lot of experience. This results in a manually driven process with a lot of expertise demanding a specialist. We recommend a distinction of a verification to prove the correctness of a system and the use of a model checker as debugging tool, since the ultimate goal is to tediously prove the system only once and not after every detection and correction of an error.

The paper is structured as follows. In Section 2, we introduce some basics about *BDDs*. Section 3 addresses traditional symbolic model checking and Section 4 its proposed enhancement with a heuristic. The automatic inference of the heuristic is the topic of Section 5. Finally, Section 6 presents our results in verifying a buggy design of the tree-arbiter and the DME.

## 17.2 BDD Basics

Ordered binary decision diagrams (*OBDDs*) introduced by Bryant [50] are a graphical representation for boolean functions. A *BDD* $G(f, \pi)$ with respect to the function $f$ and the variable ordering $\pi$ is an acyclic graph with one source and two sinks labelled with *true* and *false*. All other (internal) nodes are labelled with a boolean variable $x_i$ of $f$ and have two outgoing edges *left* and *right*. For all edges from an $x_i$ labelled node to an $x_j$ labelled node we have $\pi(i) < \pi(j)$, such that on every path in $G$ the variables are tested in the same order and at most once. Reduced *BDDs* with respect to a fixed variable ordering are a canonical representation for boolean functions. A *BDD* is reduced if isomorphic sub-*BDDs* are merged and nodes whose outgoing edges lead to the same successor are omitted. Reduced *BDDs* are build directly, integrating the reduction rules into the construction algorithm. The variable ordering $\pi$ can be choosen freely, but it has a great influence on the size of the *BDDs*, e.g. there are functions which have *BDDs* of linear size for a "good" and of exponential size for a "bad" ordering. The determination of an optimal ordering is an NP-hard problem but, for most applications, there exist several heuristics for non-optimal but "good" orderings [34]. Another method to improve the ordering is dynamic variable reordering [309] which is applied during the verification in case the *BDDs* become too large. In the following we will only speak of *BDDs*, however, we always mean reduced ordered *BDDs*.

In model checking *BDDs* help to overcome the memory limitations of explicit state representation methods [259]. They represent both sets of states and the transition relation. Model checking temporal logic properties can be reduced to the calculation of fix-points. This calculation can be performed efficiently treating lots of states in each iteration step.

An important task is to determine the set of reachable states. Starting with the set of initial states the fix-point iteration corresponds to a breadth-first search until no more new states are found. This is sufficient to check safety and simple reachability properties. To verify more complicated properties typically a backward state traversal is applied to calculate the necessary fix-points. As a drawback many unreachable states have to be represented because the reachability status of a given state is not known at the beginning of the verification.

## 17.3 Model Checking

First we expose the structure of the transition relation and examine the calculations that have to be performed to check safety properties with a classical symbolic model checker. Thereafter, we discuss alternative methods that try to overcome the weaknesses of the breadth-first search approach.

### 17.3.1 Traditional Symbolic Model Checking

In order to apply a model checker we need a description of the system and the safety property to be verified. The $\mu$-calculus is an example of a logic in which both descriptions can be expressed. The two predicates *Start* and *Goal* describe the set of initial states and the set of error states, respectively. In addition a predicate *Trans* is required that evaluates to *true* if and only if there is a transistion between two successive states. For an interleaving model the predicate *Trans* is defined by the following equation:

$$\mathit{Trans}(\mathit{State}\ s, \mathit{State}\ t) = \bigvee_{i=1}^{n} \mathit{Trans}_i(s, t_i) \wedge \mathit{CoStab}_i(s, t).$$

Systems typically consist of several interacting modules. The interaction can be synchronous, asynchronous or interleaving. Here the verification of an interleaving model is described while for the verification of the other two models only small changes in the transition relation have to be made.

The predicate $\mathit{CoStab}_i$ describes that all modules except module $i$ preserve their state and the predicate $\mathit{Trans}_i$ describes the transition relation of the single module $i$ that might depend on the states of up to all other $(n)$ modules but that only changes its own state $s_i$ into $t_i$. The state of a single module consists of several $(m)$ variables of type: bool, enumerated or integer (with limited values) or a combination of them. They are all translated into boolean variables such that for all modules we end up with an expression of the form:

$$\mathit{Trans}_i(\mathit{State}\ s, \mathit{ModuleState}\ t_i) = \bigwedge_{j=1}^{m} T_{i,j}(s, t_{i,j}),$$

where $t_{i,j}$ describes the state of variable $j$ in module $i$. The transition $T_{i,j}$ of a single variable $s_{i,j}$ describes the possibility to change its value according to its input variables or to persist in its state. A backward traversal of the system then calculates the following fix-point:

$$\mu\ F_{\mathit{Goal}}(\mathit{State}\ s).\mathit{Goal}(s) \vee (\exists \mathit{State}\ \mathit{succ}.\mathit{Trans}(s, \mathit{succ}) \wedge F_{\mathit{Goal}}(\mathit{succ})).$$

After determining the set of states satisfying this fix-point we check if it contains the initial state. As said above, the disadvantage of this approach is that many unreachable states have to be stored. The alternative is to start with the set of initial states and to make a forward traversal calculating the transitive closure *Reach* of the transition relation:

$$\mu\ \mathit{Reach}(\mathit{State}\ s).\mathit{Start}(s) \vee (\exists \mathit{State}\ prev.\mathit{Trans}(prev, s) \wedge \mathit{Reach}(prev)).$$

The efficiency can be improved: After each fix-point iteration we check if the set contains an error state in which case the verification can be aborted. Based on an interleaving (asynchronous) combination of modules, however, each order of transitions of the single modules has to be taken into account leading to state space explosion. Therefore, the sole calculation of reachable states might be impossible.

### 17.3.2 Other Approaches

An attempt to overcome the disadvantage of the unreachable states to be stored in a backward traversal is *local model checking* [67, 333]. It applies a depth-first search allowing

an intelligent choice of the next state to be expanded. It significantly reduces the verification time in case properties are checked that do not necessarily require the traversal of the whole state space. In general, local model checking uses an explicit state representation such that it cannot take profit from the elegant and space efficient representation based on *BDDs*.

An attempt to combine *partial order reduction* with *BDDs* was made in [10]. Nevertheless, it was not yet as successful as global *BDD*-model checking.

*Bounded model checking* performs symbolic model checking without *BDDs* using *SAT* decision procedures [36]. The transition relation is unrolled $k$ steps for a bounded $k$. The bound is increased until an error is found or the bound is large enough to guarantee the correctness of a successful verification. The major disadvantage of bounded model checking is the fact that it is difficult to guarantee a successful verification, since the necessary bound will be rather large and difficult to determine. Therefore, similar to our approach the most important profit of this method is a fast detection of errors.

*Validation with guided search* [352] is the only other approach known to the authors which tries to profit from a heuristic to improve model checking. The measure is the Hamming distance, i.e. the minimum number of necessary bit-flips to transfer a given bit-vector of a state to an erroneous one. For our purposes this heuristic is too weak. The lower bound presented in Section 17.5 has a larger range of values leading to a better selection of states to be expanded. Furthermore, the pure effect of the heuristic is not clearly evaluated. The authors compare the number of visited and explored states with a breadth-first search, but unfortunately the *BDD*-sizes for the state representation, the key performance measure, are not mentioned. It does not become clear which parts of the verification are performed with *BDDs* and which parts are dealt otherwise. The proposed approach is combined with two other methods: *target enlargements* and *tracks*. The former corresponds to a certain kind of bidirectional search, which is not a heuristic but a search strategy close to perimeter search [82] and the latter seems to be highly manually driven and not suitable for automatisation, one of our principal aims. The combination of their methods to find an error leads to good results, but in our opinion, it seems that the heuristic only contributes a small part to this advancement.

In contrast our algorithms entirely utilizes the *BDD* data structure such that the only interesting point are the sizes of the *BDDs* and not the number of states represented by it. In the examples of Section 17.6 the overall time and memory efficiency of our approach is shown to outperform traditional *BDD* breadth-first search.

## 17.4 Directed Model Checking

In *BDD* based breadth-first-search all states on the search horizon are expanded in one iteration step. In contrast our approach is directed by a heuristic that determines a subset of the states on the horizon to be expanded which most promisingly leads to an error state. Non-symbolic heuristic search strategies are well studied. A* [161] is an advancement of Dijkstra's algorithm [80] for determining the shortest paths between two designated states within a graph.

The additional heuristic search information helps to avoid a blind breadth-first-search traversal but still suffers from the problem that a huge amount of states has to be stored. In this section an algorithm similar to A* is proposed to improve symbolic model checking.

### 17.4.1 BDDA*

Edelkamp and Reffel have shown how *BDDs* help to solve heuristic single-agent search problems intractable for explicit state enumeration based methods [112]. The proposed algorithm *BDDA** was evaluated in the Fifteen-Puzzle and within Sokoban.

The approach exhibits a new trade-off between time and space requirements and tackles the most important problem in heuristic search, the overcoming of space limitations while avoiding a strong penalty in time. The experimental data suggests that *BDDA** challenges both breadth-first search using *BDDs* and traditional *A**. Sokoban is intractable to be solved with explicit state enumeration techniques (unless very elaborated heuristics, problem graph compressions and pruning strategies are applied) and the Fifteen-Puzzle cannot be solved with traditional symbolic search methods. It is worthwhile to note that especially in the Sokoban domain only very little problem specific knowledge has been incorporated to regain tractability.

The approach was successfully applied in AI-planning [113]. The authors propose a planner that uses *BDDs* to compactly encode sets of propositionally represented states. Using this representation, accurate reachability analysis and backward chaining are apparently be carried out without necessarily encountering exponential representation explosion. The main objectives are the interest in optimal solutions, the generality and the conciseness of the approach. The algorithm is tested against a benchmark of planning problems and lead to substantial improvements to existing solutions. The most difficult problems in the benchmark set were only solvable when additional heuristic information in form of a (fairly easy) lower bound was given.

### 17.4.2 Heuristics and A*

Let $h^*(s)$ be the length of the shortest path from $s$ to a goal state and $h(s)$ its estimate. A heuristic is called *optimistic* if it is always a lower bound for the shortest path, i.e., for all states $s$ we have $h(s) \leq h^*(s)$. It is called *consistent* if we have $h(u) \leq h(v) + 1$, with $v$ being the successor of $u$ on any solution path. Consistent heuristics are optimistic by definition and optimistic heuristics are also called *lower bounds*.

Heuristics correspond to a reweighting of the underlying problem graph. In the uniformly weighted graph we assign the following assignment to the edges $w(u, v) = 1 - h(u) + h(v)$. Fortunately, up to an additional offset the shortest paths values remain the same and no negative weighted loops are introduced. Consistent heuristics correspond to a positively weighted graph, while optimistic heuristics may lead to negative weighted edges.

In *A** there are three sets. The set *visited* of states already expanded, the set *Open* containing the states next to be expanded and the states which have not yet been encountered. During the calculation every state always belongs to exactly one of these sets. When a state is expanded it is moved from *Open* to *visited* and all its successors are moved to *Open* unless they do not already belong to *visited*. In this case they are inserted back to *Open* (*reopened*) only if the current path is shorter than the one found before. This is done until the goal state is encountered or the set *Open* is empty. In the later case there exists no path between an initial state and a goal state. The correctness result of *A** states that given an optimistic estimate the algorithm terminates with the optimal solution length.

### 17.4.3 Tailoring BDDA* for Model Checking

In the *BDD* version of *A** the set *Visited* is omitted. To preserve correctness the successors of the expanded state are always inserted into *Open*. This relates to the expansion of the entire search tree corresponding to the reweighted graph. The closely related explicit state enumeration technique is iterative deepening *A**, *IDA** for short [213]. With an increasing bound on the solution length the search tree is traversed in depth-first manner. Note, *IDA** was the first algorithm that solved the Fifteen Puzzle. The admissibility of BDDA* is inherited by the fact that Korf has shown that given an optimistic heuristic IDA* finds an optimal solution.

For model checking omitting the set *Visited* turns out not to be a good choice in general such that the option to update the set of visited states in each iteration has been reincarnated. In difference to *A**, however, the length of the minimal path to each state is not stored. The closest corresponding single-state space algorithm is *IDA** with transposition tables [305]. Transposition tables store already encountered states to determine that a given state has already been visited. This pruning strategy avoids so-called *duplicates* in the search. However it is necessary to memorize the corresponding path length to guarantee admissibility for optimistic heuristics. Fortunately one can omit this additional information when only consistent heuristics are considered. In this case the resulting cost-function obtained by the sum of path length $g$ and heuristic value $h$ is monotone.

The set *Open* is a priority queue sorted according to the costs of the states. The costs of a state $s$ is the sum of the heuristic and the number of steps necessary to reach $s$. The priority queue *Open* can be symbolically represented as a *BDD Open*(costs, state) in which the variables for the binary representation of the costs have smaller indices than those for the representation of states. In Figure 17.1 the algorithm is represented in pseudo code. The *BDD Open* corresponds to a partitioning of the states according to their costs.

Due to the variable ordering a new *BDD* operation (not included in standard *BDD* libraries) might efficiently combine three steps in the algorithm: the determination of the set of states with minimal costs contained in the queue, its costs $f_{min}$, and the new queue without these states. The function follows the path from the root node by always choosing the left successor – provided it does not directly lead to *false* – until the first state variable is encountered. This node is the root of *Min*, the persecuted path corresponds to the minimal costs $f_{min}$. The set *Open* excluding the just expanded states is obtained when *Min* is replaced by *false* probably followed by some necessary applications of the *BDD*-reduction rules.

Note, that the range of the costs has to be chosen adequately to avoid an overflow. To determine the set *Succ* the costs of the new states have to be calculated. As in *Open* only the costs of a state are stored and not the path-length the new costs are the result of the formula $f' - h' + 1 + h$ with $f'$ and $h'$ being the costs and the heuristic value of the predecessor. The value 1 is added for the effected transition and $h$ is the estimate for the new state. Afterwards it remains to update the set *Visited* which is merged with *Min*. Furthermore the new states *Succ* are added to *Open* which should contain no states comprised in *Visited*.

**Input** *BDD Start* of the initial, *BDD Goal* of the erroneous states, *BDD Trans* representing the transition relation, and *BDD Heuristic* for the estimate of the entire search space.
**Output** "Error state found!" if the algorithm succeeds in finding the erroneous state, "Complete Exploration!", otherwise.

*Visited*(State s) := *false*
*Open*(Costs f, State s) := *Start*(s) $\wedge$ *Heuristic*(f,s)
**while** ($\exists$ $s_1$,$f_1$. *Open*($f_1$,$s_1$))
  **if** ($\exists$ s′,f′. *Open*(f′,s′) $\wedge$ *Goal*(s′)) **return** "Error state found!"
  *Min*(f,s) := *Open*(f,s) $\wedge$ f=$f_{min}$
  *Succ*(f,s) := $\exists$ f′,s′,h,h′. *Heuristic*(h,s) $\wedge$ *Heuristic*(h′,s′) $\wedge$
    *Min*(f′,s′) $\wedge$ *Trans*(s′,s) $\wedge$ f=f′−h′ + 1+h
  *Visited*(s) := *Visited*(s) $\vee$ $\exists$ f. Min(f,s)
  *Open*(f,s) := (*Open*(f,s) $\vee$ *Succ*(f,s)) $\wedge$ $\neg$*Visited*(s)
**return** "Complete Exploration!"

Figure 17.1: Heuristic based algorithm in Model Checking.

## 17.5 Inferring the Heuristic

The heuristic estimates the distance (measured in the number of transition steps) from a state to an error state. According to the type of the system such a step can have different meanings. For a synchronous system one step corresponds to one step in each module. In an asynchronous system a subset of all modules can perform a step and finally for an interleaving model exactly one module executes a transition. The challenging question is how to find a lower bound estimate (optimistic heuristic) for typical systems.

First of all, $h \equiv 0$ would be a valid choice, but in this case $A*$ exactly corresponds to breadth-first-search. Therefore, the values of the heuristic have to be positive to serve as an effective guidance in the search: The more diverse the heuristic values the better the classification of states. In this case we select most promising states for failure detection and distinguish them from the rest. As an effect in each iteration only a few states have to be expanded.

The next intuitive heuristic is the Hamming distance mentioned above. The measure is optimistic if in one transition only one $x_i$ can change. Unfortunately, this is not true in general. The main drawback of this heuristic, however, is that in general the number of variables necessary to define an error state are few in comparison to the number of state variables. Hence, the Hamming distance typically has a small range of values and the number of different partitions of states are too less to significantly reduce the number of states to be expanded.

In the sequel we propose an automatic construction of a heuristic only based on the safety property and the structure of the transition function. We assume that the formula $f$ describing the error states is a boolean formula using $\wedge$ and $\vee$ while negation is only applied directly to variables. In CTL the safety property with respect to the property $f$ is denoted by AG($\neg f$).

$$Heu_f(s) = \begin{cases} \min_{k=1,\dots,n} Heu_{f_k}(s), & \text{if } f = f_1 \vee \dots \vee f_n \\ \max_{k=1,\dots,n} Heu_{f_k}(s), & \text{if } f = f_1 \wedge \dots \wedge f_n \\ Heu_{s_{i,j}}(s), & \text{if } f = s_{i,j} \\ Heu_{\overline{s_{i,j}}}(s), & \text{if } f = \overline{s_{i,j}} \end{cases}$$

Table 17.1: Property-dependent determination of heuristic values.

| $g(x_1, x_2)$ | $\text{Heu}_{s_{i,j}}(s)$ = if $(s_{i,j})$ then 0 else: |
|---|---|
| 0 | $\infty$ |
| $x_1 \wedge x_2$ | $1 + \max\{Heu_{x_1}(s), Heu_{x_2}(s)\}$ |
| $\neg(x_1 \rightarrow x_2)$ | $1 + \max\{Heu_{x1}(s), Heu_{\overline{x2}}(s)\}$ |
| $x_1$ | $1 + \text{Heu}_{x_1}(s)$ |
| $\neg(x_2 \rightarrow x_1)$ | $1 + \max\{Heu_{\overline{x1}}(s), Heu_{x_2}(s)\}$ |
| $x_2$ | $1 + Heu_{x_2}(s)$ |
| $x_1 \not\leftrightarrow x_2$ | $1 +$ (if $(x_1)$ $\min\{Heu_{\overline{x1}}(s), Heu_{\overline{x2}}(s)\}$ else $\min\{Heu_{x_1}(s), Heu_{x_2}(s)\}$) |
| $x_1 \vee x_2$ | $1 + \min\{Heu_{x_1}(s), Heu_{x_2}(s)\}$ |

Table 17.2: Transition-dependent determination of heuristic values for $t_{i,j} = g(s_{i_1,j_1}, s_{i_2,j_2})$. The remaining 8 functions are obtained by duality.

### 17.5.1 Definition

Table 17.1 describes the transformation of the formula $f$ into a heuristic $\text{Heu}_f$. In the first two cases the sub-formulas $f_k$ must not contain another $\vee$-operator (respectively $\wedge$) at the top level.

With this construction the heuristic value depends only on $\text{Heu}_{s_{i,j}}(s)$ and $\text{Heu}_{\overline{s_{i,j}}}(s)$ which rely on the structure of the transition relation. As explained in Section 17.3.1 the transition of variable $j$ in module $i$ is described by $T_{i,j}(s, t_{i,j})$. The devices $T_{i,j}$ are typically some standard electronic elements such as the logical operators *or*, *and*, *xor*, etc. In a general setting, however, they can be arbitrary formulas.

Table 17.2 exemplarily depicts the values for the function $\text{Heu}_{s_{i,j}}$ for every binary boolean formula. For a general boolean function $s_{i,j} = g : B^n \mapsto B$ with $n$ arguments the sub-function $\text{Heu}_{s_{i,j}}(s)$ has the following value:

$$\min_{x \in B^n | g(x)=1} \{ \max_{i \in \{1..n\}} \{\text{number of transitions necessary until } s_i = x_i\}\}$$

Note, that Table 17.2 is valid for both an asynchronous and a synchronous model. In the case of an interleaving model the heuristic can be improved: if $Heu_{x_1}$ and $Heu_{x_2}$ appear only once in the whole formula and $x_1$ and $x_2$ belong to different modules then *max* can be replaced by *plus*.

### 17.5.2 Refinement-Depth

Actually, the rules can be applied to $\text{Heu}_f$ infinitely often such that we have to limit the number of its applications and to define a base case.

**Definition 15** *In a first step all possible rules of Table 17.1 are applied to $Heu_f$. The* refinement-depth *of the heuristic formula is the number of times all possible replacements*

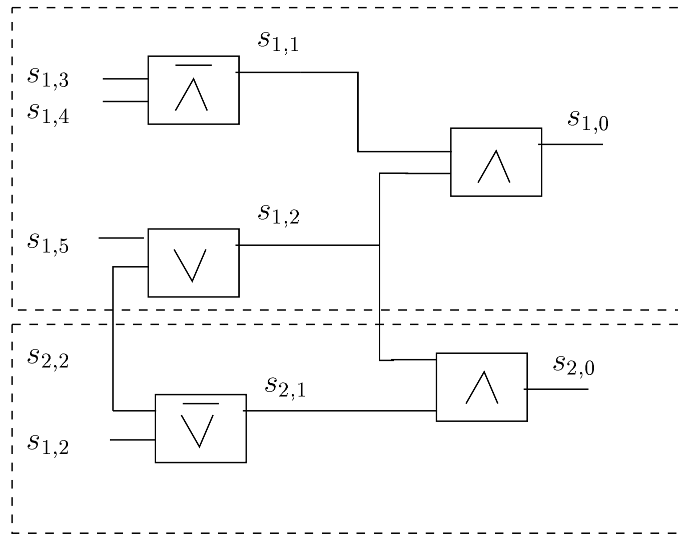


Figure 17.2: Part of an electronic circuit

*given in Table 17.2 are applied.*

The rules are applied appropriately to reach the desired depth. Afterwards each remaining $\text{Heu}_{s_{i,j}}(s)$ is replaced by

$$\text{if } (s_{i,j}) \text{ then } 0 \text{ else } 1$$

A higher refinement-depth corresponds to an improvement of the estimate, but on the other hand the *BDD* representing the heuristic becomes bigger and from a certain depth on the benefit from further refinements becomes very small or even disappears. So the aim is to find a trade-off between the *BDD*-size and the refinement-depth. In many cases already simple heuristics lead to a noticeable effect. Therefore, a feasible strategy can be to start with a simple heuristic and to refine it more and more until an error state is found.

### 17.5.3 Example

In Figure 17.2 a part of an electronic circuit is given. Let $f = s_{1,0} \wedge s_{2,0}$ be the description of the error state. Table 17.3 demonstrates the construction of the heuristic for a refinement depth of 1 and 2.

To show that our heuristic yields to a better partitioning of the state space by a wider range of heuristic values look at the following state $s = \{s_1, s_2\}$ with $s_1 = (1, 1, 1, 1, 1, 0)$ and $s_2 = (0, 0, 1, 0)$

As $s_{1,0}$ is *true* and $s_{2,0}$ is *false* the Hamming distance is 1. Our construction of the heuristic takes into account that it is not possible to reach a state where $s_{2,0} =$ *true* with one transition. Using a refinement depth of 2 leads to $\text{H}_{s_{1,0} \wedge s_{2,0}}(s) = 3$. The variable $s_{2,2}$ has to become *false* before the *nor*-element can change the value of $s_{2,1}$ and in the third transition $s_{2,0}$ can switch to *true*.

### 17.5.4 Properties

As indicated the heuristic can be improved to allow a better partitioning of the set of states to be expanded. A non-optimistic heuristic can lead to a faster detection of an

| $x$ | *Heu*$_x(s)$ for $x$ =*true* in $s$ | *Heu*$_x(s)$ for $x$ =*false* in $s$ |
|---|---|---|
| For refinement depth 1: | | |
| $s_{1,0}$ | 0 | $1 + \max\{\mathrm{H}_{s_{1,1}}, \mathrm{H}_{s_{1,2}}\}$ |
| $s_{2,0}$ | 0 | $1 + \max\{\mathrm{H}_{s_{1,2}}, \mathrm{H}_{s_{2,1}}\}$ |
| For refinement depth 2: | | |
| $s_{1,1}$ | 0 | $1 + \min\{\mathrm{H}_{\overline{s_{1,3}}}, \mathrm{H}_{\overline{s_{1,4}}}\}$ |
| $s_{1,2}$ | 0 | $1 + \min\{\mathrm{H}_{s_{1,5}}, \mathrm{H}_{s_{2,2}}\}$ |
| $s_{2,1}$ | 0 | $1 + \max\{\mathrm{H}_{\overline{s_{2,2}}}, \mathrm{H}_{\overline{s_{2,3}}}\}$ |
| Base cases: | | |
| $s_{1,5}, \overline{s_{1,3}}, \overline{s_{1,4}}$ | 0 | 1 |
| $s_{2,2}, \overline{s_{2,2}}, \overline{s_{2,3}}$ | 0 | 1 |

Table 17.3: Example of the heuristic estimate.

erroneous state but on the other hand it can increase the length of the counter example. The construction of the heuristic however always leads to an optimistic heuristic. To prove this we will use the following lemma:

**Lemma 14** $\mathrm{Heu}_{s_{i,j}}(s)$ *is a lower bound for the number of transitions which are necessary to reach a state from* $s$ *where* $s_{i,j} = true$.

**Proof:** The property will be shown by induction on the refinement depth.

**Refinement depth 0:** In this case we have *Heu*$_{s_{i,j}}$ $=$ if $(s_{i,j})$ then 0 else 1. (The argumentation for negated variables $\overline{s_{i,j}}$ is similar.) If $s_{i,j}$ is true no transition step is necessary. Hence, *Heu*$_{s_{i,j}}(s) = 0$. In case $s_{i,j}$ is false at least one transition step has to be made to change its value. Therefore, *Heu*$_{s_{i,j}}(s) = 1$ is a lower bound.

**Refinement depth $k$:** We will suppose that for a refinement depth less than $k$ the functions *Heu*$_{s_{i,j}}$ fulfill the desired property. After the first application of a rule from Table 17.2 for the introduced formulae *Heu*$_{x_i}$ we have a refinement depth of $k-1$. This implies that it takes at least *Heu*$_{x_i}$ steps to change the values of the involved variables. It depends on the formula $g(x_1, x_2)$ which changes the value of $s_{i,j}$ that it is necessary for both variables $x_1$ and $x_2$ to have a certain value, or that it is sufficient that one of them has a certain value. This determines if the minimum or the maximum of the involved functions *Heu*$_{x_i}$ is used. In both cases after the variables $x_1$ and $x_2$ have been assigned to a value, which allows $g$ to change the value of $s_{i,j}$ from *false* to *true*, at least one additional step is necessary. Therefore, 1 can be added and *Heu*$_{s_{i,j}}$ still remains a lower bound.

■

Using Lemma 14 it it quite easy to prove that *Heu*$_f$ according to the construction introduced above is a lower bound estimate.

**Theorem 17** *The function* $\mathrm{Heu}_f$ *is optimistic.*

**Proof:**
It remains to show that the rules of Table 17.1 lead to an optimistic heuristic since the sub-functions $Heu_{s_{i,j}}$ underestimate the number of transitions necessary to achieve the desired value for $s_{i,j}$. This can be shown easily by induction on the number of applications of the rules of Table 17.1 so we will only explain the main idea for the proof.

It is based on the fact that for an $\vee$-formula it is sufficient that one of the $f_i$ becomes true, so the minimum of the $Heu_{f_i}$ is chosen and for an $\wedge$-formula all $f_i$ have to be fulfilled so the maximum of the $Heu_{f_i}$ can be chosen for the heuristic value. ■

Note, that for an asynchronous or a synchronous system in one transition step the values of various variables can change, therefore it is not possible to summarize over the $Heu_{f_i}$ for example in case of an $\wedge$-formula. In contrast, for an interleaving model the sum could be used if the $f_i$ depend on variables in different modules because only one module can change its state in a single transition.

As already indicated to guarantee the computation of the minimal counter-example in the proposed extension to *BDDA** it is not sufficient to use an optimistic heuristic. Fortunately, it is possible to show the consistency of our automatically constructed heuristic:

**Theorem 18** *The function $Heu_f$ is consistent.*

**Proof:**
We have to show that

$$\forall \mathit{State}\ s, t. \mathit{Trans}(s,t) \Rightarrow \mathit{Heu}_f(s) \leq 1 + \mathit{Heu}_f(t).$$

This property follows directly from the fact that for all variables $x$ we have

$$\forall \mathit{State}\ s, t. \mathit{Trans}(s,t) \Rightarrow \mathit{Heu}_x(s) \leq 1 + \mathit{Heu}_x(t).$$

We will prove this by induction on the refinement depth similar to Lemma 14. For a refinement depth of 0 there is nothing to prove because $\mathit{Heu}_x(s) \leq 1$. For refinement depth $k$ we will show the property for the operator $\wedge$ (cf. Table 17.2). In this case $\mathit{Heu}_x(s)$ is defined as $1 + \max\{\mathit{Heu}_{x_1}(s), \mathit{Heu}_{x_2}(s)\}$. For $\mathit{Heu}_{x_1}$ and $\mathit{Heu}_{x_2}$ we have a refinement depth of $k-1$ so the property holds for these formulas:

$$\begin{aligned} \mathit{Heu}_x(s) &\leq 1 + \max\{1 + \mathit{Heu}_{x_1}(t), 1 + \mathit{Heu}_{x_2}(t)\} \\ &\leq 1 + (1 + \max\{\mathit{Heu}_{x_1}(t), \mathit{Heu}_{x_2}(t)\}) \\ &\leq 1 + \mathit{Heu}_x(t) \end{aligned}$$

The proof for the other operators of Table 17.2 is analogue expect for the operator $\not\leftrightarrow$. The interesting case is a transition where in state $s$ the variables $x_1$ and $x_2$ are assigned to *true* and in state $t$ both variables are assigned to *false*. In this case the structure of the formula changes: For state $s$ we have

$$\mathit{Heu}_x(s) = 1 + \min\{\mathit{Heu}_{\overline{x_1}}(s), \mathit{Heu}_{\overline{x_2}}(s)\}$$

and in state $t$ we establish

$$Heu_x(t) = 1 + \min\{Heu_{x_1}(t), Heu_{x_2}(t)\}.$$

The circumstance that there is a transition from $s$ to $t$ which changes the values of $x_1$ and $x_2$ from *true* to *false* implies that $Heu_{\overline{x1}}(s) = Heu_{\overline{x2}}(s) = 1$ while for state $t$ we have $Heu_{x_1}(t)$, $Heu_{x_2}(t) \geq 1$. Therefore, the following equation completes the proof:

$$Heu_x(t) = 1 + \min\{Heu_{x_1}(t), Heu_{x_2}(t)\} \geq 1 + \min\{1, 1\} = 2 = Heu(s)$$

■

Note, that breadth-first-search finds the error state in the minimal number of iterations. In contrast in the heuristic search approach several states remain unexpanded in each iteration such that the number of necessary iteration steps increases. In the worst case we have a quadratic growth in the number of iterations [112]. On the other hand, especially for large systems, a transition step expanding only a small subset of the states is much faster than a transition based on all states. Therefore, this apparent disadvantage even turns out to be very time-efficient surplus as the examples in the next section will show.

## 17.6 Experimental Results

In our experiments we used the $\mu$-calculus model checker $\mu$cke [35] which accepts the full $\mu$-calculus for its input language [283]. The *while*-loop has to be converted into a least fixpoint. As it is not possible to change two sets (*Open, Visited*) in the body of one fixpoint the *Visited* set is simulated by one slot in the *BDD* for *Open*. The next problem is that the function for *Open* is not monotone because states are deleted from it after they have been expanded. Monotony is a sufficient criterion to guarantee the existence of fixpoints. The function for *Open* is not a syntactic correct $\mu$-calculus formula but as the termination of the algorithm is guaranteed by the monotony of the set *Visited* we can apply the standard algorithm for the calculation of $\mu$-calculus fixpoints.

Unfortunately, we cannot take advantage of the special *BDD* operation determining the set of states with minimal costs in this case. These calculations have to be simulated by standard operations leading to some unnecessary overhead that in the visible future has to be avoided in a customized implementation.

For the evaluation of our approach we use the example of the tree-arbiter [81] a mechanism for distributed mutual exclusion: $2n$ user want to use a resource which is available only once and the tree-arbiter manages the requests and acknowledges avoiding a simultaneous access of two different users. The tree-arbiter consists of $2n - 1$ modules of the same structure such that it is very easy to scale the example. Since we focus on error detection we experiment with an earlier incorrect version – also published in [81] – using an interleaving model.

The heuristic was devised according to the description in Section 17.5 with a refinement-depth of 6. We also experimented with larger depths which implied a reduction neither in time nor in size. Since the algorithm for the automatic construction of the heuristic has not yet been implemented and since the number of different errors increases very fast with the size of the tree-arbiter we searched for the detection of a special error

| # Mod | BFS | | | Heuristic | | | |
|---|---|---|---|---|---|---|---|
| | #it | max nodes | time | depth | #it | max nodes | time |
| 15 | 30 | 991,374 | 46s | 4 | 104 | 10,472,785 | 483s |
| | | | | 6 | 127 | 5,715,484 | 288s |
| 17 | 42 | 18,937,458 | 3,912s | 6 | 157 | 7,954,251 | 476s |
| 19 | 44 | 22,461,024 | 6,047s | 6 | 157 | 8,789,341 | 540s |
| 21 | 44 | 26,843,514 | 24,626s(9) | 6 | 157 | 9,097,823 | 530s |
| 23 | >40 | - | >17,000s | 6 | 157 | 9,548,269 | 516s |
| 25 | - | - | - | 6 | 169 | 21,561,058 | 1,370s |
| 27 | - | - | - | 6 | 169 | 25,165,795 | 1,818s(1) |
| | | | | 6(x2) | 593 | 23,798,202 | 1,970s |

Table 17.4: Results for the tree-arbiter. In parenthesis the number of garbage collections is given.

case. Table 17.4 shows the results in comparison with a classical forward breadth first search. To guarantee the fairness of the comparison we terminated the search at the time the error state has first been encountered.

For the tree-arbiter with 15 modules or less the traditional approach is faster and less memory consuming, but for larger systems its time and memory efficiency decreases very fast. On the other hand, the heuristic approach found the error even in large systems, since its memory and time requirements increases slowly. For the tree-arbiter with 23 modules the error could not be found with breadth-first-search and already for the version with 21 modules 9 garbage collections were necessary not to exceed the memory limitations, whereas the first garbage collection with the heuristic method was invoked at a system of 27 modules. For the tree-arbiter with 27 modules we also experimented with the heuristic. When we double its values the heuristic fails to be optimistic, but the error detection becomes available without any garbage collection. Moreover, although more than three times more iterations were necessary only about $8\%$ more time was consumed. This illustrates that there is much room for further research in refinements to the heuristic.

The second example we used for the evaluation of our approach is the asynchronous DME [81]. Like the tree-arbiter it consists of $n$ identical modules and it is also a mechanism for distributed mutual exclusion. The modules are arranged in a ring structure whereas the modules of the tree-arbiter form a pyramid. In this case we also experimented with the set *visited* and it turns out that it was more efficient to omit it like proposed in [112]. For this variation only a small change in the calculation of *Open* is necessary. Like in the previous example the results in Table 17.5 show that the heuristic approach is more memory efficient and less time-consuming. The first experiment in the Table uses the set *visited* that was omitted in the other experiments. This led to a greater iteration depth because several states are visited more than once. Nevertheless this turned out to be more time and memory efficient. The increase of the refinement-depth to 7 allows to reduce the verification time and no garbage collection remains necessary.

## 17.7 Conclusion and Discussion

We presented how a heuristic can successfully be integrated into symbolic model checking. It is recommended to distinguish between the use of a model checker in order to

| # Mod | BFS | | | Heuristic | | | |
|---|---|---|---|---|---|---|---|
| | #it | max nodes | time | depth | #it | max nodes | time |
| 6 | 23 | 26,843,514 | 5,864s(5) | 6v | 35 | 29,036,025 | 2,207s(4) |
| | | | | 6 | 53 | 25,165,795 | 1,009s(1) |
| | | | | 7 | 53 | 25,159,862 | 813s(0) |

Table 17.5: Results for the asynchronous DME. In parenthesis the number of garbage collections is given.

prove a property and the use as a debugging tool. For debugging exhaustive search of the reachable state space can be avoided and the heuristic can decrease both the number of expanded states and the *BDD*-sizes which allows the treatment of bigger systems. It was shown how a heuristic can be automatically designed for a large class of systems allowing the application of this method also for non-experts.

The experiments demonstrated the effectiveness of the approach and we plan to test the algorithm with more example data and to evaluate further refinements of the heuristic and its construction.

There are lots of choices for an experienced user to modify and improve the estimate or even to use non-optimistic heuristics allowing a better partitioning of the state space. This can be more important than the determination of the minimal counter-example. Pearl [287] discusses limits and possibilities of overestimations in corresponding explicit search algorithms. One proposed search scheme, called *WIDA**, considers costs of the form $f = \alpha g + (1 - \alpha)h$. If $\alpha \in [.5, 1]$ the algorithm is admissible. In case $\alpha \in [0, .5)$ the algorithm searches according to overestimations of the heuristic value compared to the path length $g$. The literature clearly lacks theoretical and practical results for symbolic searches according to non-optimistic heuristics.

# Part VII

# Route Planning

# Paper 18

# Localizing A*

Stefan Edelkamp
Institut für Informatik
Am Flughafen 17
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

Stefan Schrödl
DaimlerChrysler Research and Technology
1510 Page Mill Road
Palo Alto, CA 94303
schroedl@rtna.daimlerchrysler.com

In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.

**Abstract**

Heuristic search in large problem spaces inherently calls for algorithms capable of running under restricted memory. This question has been investigated in a number of articles. However, in general the efficient usage of two-layered storage systems is not further discussed. Even if hard-disk capacity is sufficient for the problem instance at hand, the limitation of *main memory* may still represent the bottleneck for their practical applications. Since breadth-first and best-first strategies do not exhibit any locality of expansion, standard *virtual memory management* can soon result in thrashing due to excessive page faults.

In this paper we propose a new search algorithm and suitable data structures in order to minimize page faults by a local reordering of the sequence of expansions. We prove its correctness and completeness and evaluate it in a real-world scenario of searching a large road map in a commercial route planning system.

## 18.1 Introduction

Heuristic search algorithms are usually applied to huge problem spaces. Hence, having to cope with memory limitations is an ubiquitous issue in this domain. Since the development of the *A** algorithm [161], the main objective has always been to develop methods to regain tractability.

The class of *memory-restricted search algorithms* has been developed under this aim. The framework imposes an absolute upper bound on the total memory the algorithm may use, regardless of the size of the problem space. Most papers do not explicitly distinguish whether this limit refers to disk space or to working memory, but frequently the latter one appears to be implicitly assumed.

*IDA** explores the search space by iterative deepening and uses space linear in the solution length, but may revisit the same node again and again [213]. It does not use additionally available memory. *MREC* switches from *A** to *IDA** if the memory limit is reached [324]. In contrast, *SMA** [311] reassigns the space by dynamically deleting a previously expanded node, propagating up computed $f$-values to the parents in order to save re-computation as far as possible. Eckerle and Schuierer improve the dynamic re-balancing of the search tree [90]. However, it remains to be shown that these algorithms in general outperform *A** or *IDA** since they impose a large administration overhead. A more recent work employs stochastic node caching and is shown to reduce the number of visited nodes compared to *MREC* [265].

Even if secondary storage is sufficient, limitation of *working memory* may still represent a bottleneck for practical applications. Modern operating systems provide a general-purpose mechanism for processing data larger than available main memory called *virtual memory*. Transparently to the program, *swapping* moves parts of the data back and forth from disk as needed. Usually, the virtual address space is divided up into units called *pages*; the corresponding equal-sized units in physical memory are called *page frames*. A page table maps the virtual addresses on the page frames and keeps track of their status (loaded/absent). When a *page fault* occurs, i.e., a program tries to use an unmapped page, the CPU is interrupted; the operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the referenced page into the page frame just freed, changes the map, and restarts the trapped instruction. In modern computers memory management is implemented on hardware with a page size commonly fixed at 4096 Byte.

Various *paging strategiespaging strategy* have been explored that aim at minimizing page-faults. Belady has shown that an optimal off-line page exchange strategy deletes the page, which will not be used for the longest time [26]. Unfortunately, the system, unlike possibly the application program itself, cannot know this in advance. Several different on-line algorithms for the paging problem have been proposed, such as *Last-In-First-Out (LIFO)*, *First-In-First-Out (FIFO)*, *Least-Recently-Used (LRU)*, *Least-Frequently-Used (LFU)*, *Flush-When-Full (FWF)*, etc. [336]. Sleator and Tarjan proved that *LRU* is the best on-line algorithm for the problem achieving an optimal competitive ratio equal to the number of pages that fit into main memory [329].

Programmers can reduce the number of page faults by designing data structures that exhibit *memory locality*, such that successive operations tend to access nearby memory addresses. However, sometimes it would be desirable to have more explicit control of secondary memory manipulations. For example, fetching data structures larger than the

system page size may require multiple disk operations. A file buffer can be regarded as a kind of *"software" paging* that mimics swapping on a coarser level of granularity. Generally, an application can outperform the operating system's memory management because it is well-informed to predict future memory access.

Particularly for search algorithms, system paging can become the major bottleneck. We experienced this problem when applying $A*$ to the domain of route planning. Node structures become large, compared to hardware pages; moreover, $A*$ does not respect locality at all; it explores nodes in the strict order of $f$ values, regardless of their neighborhood, and hence jumps back and forth in a spatially unrelated way for only marginal differences in the estimation value.

In the following we present a new heuristic search algorithm to overcome this lack of locality. In connection with software paging strategies, it can lead to a significant speedup. The idea is to organize the graph structure for spatial locality and to expand spatial local data even if it can lead to a possible non-optimal solution. As a consequence, the algorithm cannot stop with the first solution found, but has to do the additional work of exploring all pending paths. However, the increased number of node expansions can be outweighed by the reduction in the number of page faults.

In the next section, we review traditional $A*$ and extend it so as to allow for node expansions in arbitrary order. We prove its correctness and completeness, and as a byproduct we fix a minor lack of accuracy in the traditional proof for $A*$. Then, we describe a data structure called *Heap-Of-Heaps* that is suitable to accommodate locality and is based on a partitioning of the search space. Finally the algorithm is evaluated within a commercial route planning system.

## 18.2 The Algorithm

We start by characterizing the standard $A*$ algorithm [161] in an unusual but concise way on the basis of Dijkstra's algorithm to find shortest paths in (positively) weighted graphs from a *start node* $s$ to a set of *goal nodes* $T$ [80]. Dijkstra's algorithm uses a priority queue *Open* maintaining the set of currently reached yet unexplored nodes. If $f(u)$ denotes the total weight of the currently best explored path from $s$ to some node $u$ (also called the *merit* of $u$), the algorithm always selects a node from *Open* with minimum $f$ value for expansion, updates its successors' $f$-values, and transfers it to the set *Closed* with established minimum cost path.

### 18.2.1 Traditional $A^*$ = Dijkstra + Re-weighting

Algorithm $A*$ accommodates the information of a *heuristic* $h(u)$, which estimates the minimum cost of a path from node $u$ to a goal node in $T$. It can be cast as a search through a re-weighted graph. More precisely, the edge weights $w$ are replaced by new weights $\hat{w}$ by adding the heuristic difference: $\hat{w}(u,v) = w(u,v) - h(u) + h(v)$. At each instant of time in the re-weighted Dijkstra algorithm, the merit $f$ of a node $u$ is the sum of the new weights along the currently cheapest path explored by the algorithm.

By this transformation, negative weights can be introduced. Nodes that have already been expanded might be encountered on a shorter path. Thus, contrary to Dijkstra's algorithm, $A*$ deals with them by possibly re-inserting nodes from *Closed* into *Open*.

On every path $p$ from $s$ to $u$ the accumulated weights in the two graph structures differ by $h(s)$ and $h(u)$ only, i.e., $w(p) = \hat{w}(p) - h(u) + h(s)$. Consequently, on every cycle $c$ we have $\hat{w}(c) = w(c) \geq 0$, i.e., the re-weighting cannot lead to negatively weighted cycles so that the problem remains solvable.

Let $\delta(u, v)$ and $\hat{\delta}(u, v)$ denote the least-cost path weights between nodes $u$ and $v$ in the initial resp. re-weighted graphs. The heuristic $h$ is called *consistent* if and only if $\hat{w}(u, v) \geq 0$ for all $u$ and $v$. It is called *optimistic* if $h(u) \leq \min\{\delta(u, t) | t \in T\} = h^*(u)$. This is equivalent to the condition $\min\{\hat{\delta}(u, t) | t \in T\} \geq 0$.

For convenience, since in the following we are dealing only with the transformed weights, we will write $w$ instead of $\hat{w}$.

### 18.2.2 Invariance Condition

In each iteration of the *A** algorithm, the element $u$ with minimum $f$ value is chosen from the set *Open* and is inserted into *Closed*. Then the set of successors $\Gamma(u)$ is generated. Each node $v \in \Gamma(u)$ is inspected and *Open* and *Closed* are adjusted according to the following procedure *Improve*.

**Procedure** *Improve* (*Node* $u$, *Node* $v$)
    **if** ($v \in$ *Open*)
        **if** $(f(u) + w(u, v) < f(v))$
            *Open.DecreaseKey*$(v, f(u) + w(u, v))$
        **else if** ($v \in$ *Closed*)
            **if** $(f(u) + w(u, v) < f(v))$
                *Closed.Delete*$(v)$
                *Open.Insert*$(v, f(u) + w(u, v))$
    **else**
        *Open.Insert*$(v, f(u) + w(u, v))$

The core of the standard optimality proof of *A** published in AI-literature [287] consists of an invariance stating that while the algorithm is running there is always a node $v$ in the *Open* list on an optimal path with the optimal $f$-value $f(v) = \delta(s, v)$. In our opinion, this reasoning is true but lacks some formal rigidness: if the child of a node with optimal $f$-value was already contained in *Closed* (be it with optimal $f$ value), then it wouldn't be reopened and the invariance would be violated. It is part of the proof to show that this situation cannot occur. Thus, we strengthen the invariance condition by requiring the node not to be followed by any *Closed* node on the same optimal solution path.

**Invariance I.** *Let* $p = (s = v_0, \ldots, v_n = t)$ *be a least-cost path from the start node* $s$ *to a a goal node* $t \in T$. *Application of* Improve *preserves the following invariance: Unless* $v_n$ *is in* Closed *with* $f(v_n) = \delta(s, v_n)$, *there is a node* $v_i$ *in* Open *such that* $f(v_i) = \delta(s, v_i)$, *and no* $j > i$ *exists such that* $v_j$ *is in* Closed *with* $f(v_j) = \delta(s, v_j)$.

**Proof:** W.l.o.g. let $i$ be maximal among the nodes satisfying (I). We distinguish the following cases:

1. Node $u$ is not on $p$ or $f(u) > \delta(s, u)$. Then node $v_i \neq u$ remains in *Open*. Since no $v$ in *Open* $\cap$ $p$ $\cap \Gamma(u)$ with $f(v) = \delta(s, v) \leq f(u) + w(u, v)$ is changed and no other node is added to *Closed*, (I) is preserved.

2. Node $u$ is on $p$ and $f(u) = \delta(s,u)$. If $u = v_n$, there is nothing to show.

   First assume $u = v_i$. Then *Improve* will be called for $v = v_{i+1} \in \Gamma(u)$; for all other nodes in $\Gamma(u) \setminus \{v_{i+1}\}$, the argument of case 1 holds. According to (I), if $v$ is in *Closed*, then $f(v) > \delta(s,v)$, and it will be reinserted into *Open* with $f(v) = \delta(s,u) + w(u,v) = \delta(s,v)$. If $v$ is neither in *Open* or *Closed*, it is inserted into *Open* with this merit. Otherwise, the *DecreaseKey* operation will set it to $\delta(s,v)$. In either case, $v$ guarantees the invariance (I).

   Now suppose $u \neq v_i$. By the maximality assumption of $i$ we have $u = v_k$ with $k < i$. If $v = v_i$, no *DecreaseKey* operation can change it because $v_i$ already has optimal merit $f(v) = \delta(s,u) + w(u,v) = \delta(s,v)$. Otherwise, $v_i$ remains in *Open* with unchanged $f$-value and no other node besides $u$ is inserted into *Closed*; thus, $v_i$ still preserves (I).

■

Note that we have not required $f$ to be optimistic. Under this assumption, the *optimality* of *A** is implied as a corollary, i.e., the fact that a solution returned by the algorithm is indeed a shortest one. To see this, suppose that the algorithm terminates the search process with the first node $t'$ in the set of goal nodes $T$ and $f(t')$ is not optimal. Then $f(t') > \delta(s,u) + \min\{\delta(u,t) | t \in T\} \geq \delta(s,u) = f(u)$, since for an optimistic estimate the value $\min\{\delta(u,t) | \, t \in T\}$ is not negative. This contradicts the choice of $t'$.

### 18.2.3 General-Node-Ordering $A^*$

*Move ordering* is a search optimization technique which has been explored in depth in the domain of two-player games and single-agent applications. It is well-known that substituting the priority queue by a stack or a FIFO-queue results in a depth-first resp. breadth-first traversal of the problem graph. In this case the *DeleteMin* operation is replaced by *Pop* or *Dequeue*, respectively. In the following we will assume a generic operation *DeleteSome* not imposing any restrictions on the selection criteria. The subsequent section will give an implementation that is allowed to select nodes which are "local" to to previously expanded nodes with respect to the application-dependent storage scheme, even though they do not have a minimum $f$ value.

```
Function General-Node-Ordering A*
    Open.Insert(s, h(s))
    α ← ∞; bestSolution ← ∅
    while not (Open.IsEmpty())
        u ← Open.DeleteSome()
        Closed.Insert(u)
(*)     if (f(u) > α) continue
        if (u ∈ T ∧ f(u) < α)
            α ← f(u); bestSolution ← retrieved path to u
        else Γ(u) ← Expand(u)
            for all v in Γ(u)
                Improve(u, v)
    return bestSolution
```

In contrast to *A**, reaching the first goal node will no longer guarantee optimality of the found solution path. Hence, the algorithm has to continue until the *Open* list runs empty. By storing and updating the current best solution path length as a global lower bound value $\alpha$, we give an anytime extension to A* that improves the solution quality over time. The concept can be compared to the linear best first algorithm *Depth-First-Branch-and-Bound* [214].

**Theorem 19** *If the heuristic estimate $h$ is* optimistic*,* General-Node-Ordering A* *is optimal.*

**Proof:** Upon termination, each node inserted into *Open* must have been selected at least once. Suppose that invariance (I) is preserved in each loop, i.e., that there is always a node $v$ in the *Open* list on an optimal path with $f(v) = \delta(s, v)$. Thus the algorithm cannot terminate without eventually selecting the goal node on this path, and since by definition it is not more expensive than any found solution path and *bestSolution* maintains the currently shortest path, an optimal solution will be returned. It remains to show that the invariance (I) holds in each iteration. If the extracted node $u$ is not equal to $v$ there is nothing to show. Otherwise $f(u) = \delta(s, u)$. The bound $\alpha$ denotes the currently best solution length. If $f(u) \leq \alpha$ the condition in (*) is not fulfilled and no pruning takes place. On the other hand $f(u) > \alpha$ leads to a contradiction since $\alpha \geq \delta(s, u) + \min\{\delta(u, t) | t \in T\} \geq \delta(s, u) = f(u)$ (the latter inequality is justified by $h$ being optimistic). ∎

**Theorem 20** *Algorithm* General-Node-Ordering A* *is complete, i.e., terminates on finite graphs.*

**Proof:** For each successor generation, *General-Node-Ordering A** adds new links to its traversal tree. Moreover, the algorithm only reopens a node in *Closed* when it finds a *strictly* cheaper path to it and, as said above, re-weighting of positively weighted graphs keeps weights of cycles positive. Hence, the algorithm considers at most the number of acyclic path of the underlying finite graph. This number is finite and, therefore, the algorithm terminates. ∎

## 18.3 The Heap-Of-Heaps Data Structures

Let us briefly review the usual A* implementation in terms of data structures. The set *Open* is realized as a priority queue (heap) supporting the operations *IsEmpty*, *Min*, *Insert*, *DecreaseKey DeleteMin*. The membership tests $v \in$ *Open* resp. $v \in$ *Closed* in procedure *Improve* are implemented using a hash table $T$. This makes explicit storage of the *Closed* set obsolete, since it is equal to $T \setminus$ *Open*.

For large node structures, it is inefficient to move them physically around; rather, they are maintained in an auxiliary data structure $D$ containing all graph information. $D$ can also contain the links related to the heap and to the hashing chains maximizing *memory locality* with respect to node operations. If the graph is entirely stored, the hash table collapses with $D$. In some cases there is even no other option than explicit storage, e.g. in the domain of route planning.

Our approach to achieve memory locality is to find a suitable partition of the search space and of all associated data structures into a set of (software) pages $P_1, \ldots, P_k$. We assume a function $\phi$ : *Node* $\to \{1, \ldots, k\}$ which maps each node to the corresponding page it is contained in.

The data structure *Heap-Of-Heaps* represents the *Open* set. It consists of a collection of $k$ priority queues $H_1, \ldots, H_k$, one for each page. At any instant, one of the heaps, $H_{active}$, is designated as being *active*. One additional priority queue $\mathcal{H}$ keeps track of the root nodes of all $H_i$ with $i \neq$ *active*; It is used to quickly find the overall minimum across all of these heaps.

The following operations are delegated to the member priority queues $H_i$ in the straightforward way. Whenever necessary, $\mathcal{H}$ is updated accordingly.

**Function** *IsEmpty*()
    **return** $\bigwedge_{i=1}^{k} H_i$.*IsEmpty*()

**Procedure** *Insert*(*Node* $u$, *Merit* $f(u)$)
    **if** ($\phi(u) \neq$ *active* $\wedge$ $f(u) < f(H_{\phi(u)}$.*Min*()))
        $\mathcal{H}$.*DecreaseKey*($H_{\phi(u)}, f(u)$)
    $H_{\phi(u)}$.*Insert*($u$, $f(u)$)

**Procedure** *DecreaseKey*(*Node* $u$, *Merit* $f(u)$)
    **if** ($\phi(u) \neq$ *active* $\wedge$ $f(u) < f(H_{\phi(u)}$.*Min*()))
        $\mathcal{H}$.*DecreaseKey*($H_{\phi(u)}, f(u)$)
    $H_{\phi(u)}$.*DecreaseKey*($u$,$f(u)$)

Operation *DeleteSome* performs *DeleteMin* on the active heap.

**Function** *DeleteSome*()
    *CheckActive*()
    **return** $H_{active}$.*DeleteMin*()

The *Insert* and *DecreaseKey* operations can affect all heaps. However, the hope is that the number of adjacent pages of the active page is small and that they are already in memory or have to be loaded only once; all other pages and priority queues remain unchanged and do not have to reside in main memory.

As the aim is to minimize the number of switches between pages, the algorithm favors the *active* page by continuing to expand its nodes although the minimum $f$ value might already exceed the minimum of all remaining priority queues. There are two control parameters: An *activeness bonus* $\Delta$ and an estimate $\Lambda$ for the cost of an optimum solution.

**Procedure** *CheckActive*()
    **if** ($H_{active}$.*IsEmpty*() $\vee$
        $(f(H_{active}$.*Min*()) $- f(\mathcal{H}$.*Min*().*Min*()) $> \Delta$
            $\wedge\ f(H_{active}$.*Min*()) $> \Lambda$))
        $\mathcal{H}$.*Insert*($H_{active}, f(H_{active}$.*Min*()))
        $H_{active} \leftarrow \mathcal{H}$.*Min*()
        $\mathcal{H}$.*Remove*($H_{active}$)

If the minimum $f$-value of the active heap is larger than that of the remaining heaps plus the *activeness bonus* $\Delta$, the algorithm may switch to the priority queue satisfying the minimum root $f$ value. Thus, $\Delta$ discourages page switches by determining the proportion of a page to be explored. As it increases to large values, in the limit each activated page is searched to completion.

However the active page still remains valid, unless $\Lambda$ is exceeded. The rationale behind this second heuristic is that one can often provide a heuristic for the total least cost path which is, on the average, more accurate than that obtained from $h$, but which might be overestimating in some cases.

With this implementation, algorithm *General-Node-Ordering A** itself remains almost unchanged, i.e., the data structure and page handling is transparent to the algorithm. Traditional A* arises as a special case for $\Delta = 0$ and $\Lambda < h^*(s)$, where $h^*(s)$ denotes the actual minimum cost between the start node and a goal node.

Optimality is guaranteed, since we leave the heuristic estimates unaffected by the heap prioritization scheme, and since each node inserted into the *Heap-Of-Heaps* structure is eventually returned by *DeleteMin*.

## 18.4 Experiments

In our experiments we incorporated our algorithm into a commercially available route planning system running on Windows platforms. The system covers an area of approximately 800 $\times$ 400 km at a high level of detail, and comprises approximately 910,000 nodes (road junctions) linked by 2,500,000 edges (road elements). The entire graph structure, together with the members needed for the search algorithm, results in a total memory size of 40 MByte, which already exceeds the advertized minimum main memory hardware requirement of 32 MByte.

For long-distance routes, conventional $A^*$ expands the nodes in a spatially uncorrelated way, jumping to a node as far apart as some 100 km, but possibly returning to the successor of the previous one in the next step. Therefore, the working set gets extremely large, and the virtual memory management of the operating system leads to excessive paging and is the main burden on the computation time.

As a remedy, we achieve memory locality of the search algorithm by exploiting the underlying spatial relation of connected nodes. Nodes are geographically sorted according to their coordinates in such a way that neighboring nodes also tend to appear close to each other. A page consists of a constant number of successive nodes (together with the outgoing edges) according to this order. Thus, pages in densely populated regions tend to cover a smaller area than those representing rural regions. For not too small sizes, the connectivity within a page will be high, and only a comparably low fraction of road elements cross the boundaries to adjacent pages. Fig. 18.1 shows some bounding rectangles of nodes belonging to the same page.

There are three parameters controlling the behavior of the algorithm with respect to secondary memory, the algorithm parameters $\Delta$ and $\Lambda$, and the (software) page size. The latter one should be adjusted so that the active page and its adjacent pages together roughly fit into available main memory. The optimum solution estimate $\Lambda$ is obtained by calculating the Euclidean distance between the start and the goal and adding a fixed percentage.

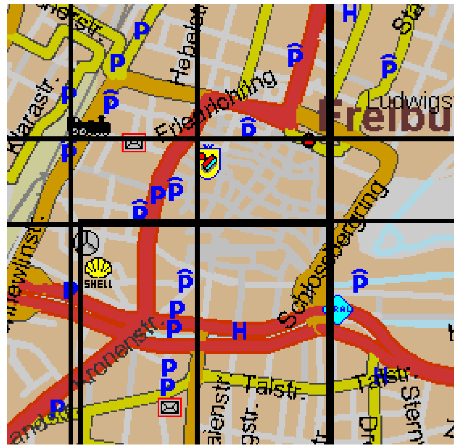


Figure 18.1: The granularity of the partition (lines indicate bounding rectangles of pages).

Fig. 18.2 opposes the number of page faults to the number of node expansions for varying page size and $\Delta$. We observe that the rapid decrease of page faults compensates the increase of expansions (note the logarithmic scale). Using an activeness bonus of about 2 km suffices to decrease the value by more than one magnitude for all page sizes. At the same time the number of expanded nodes increases by less than ten percent.

Fig. 18.3 depicts the corresponding influence of $\Lambda$. In this case the reduction of page faults by more than a magnitude can be achieved by investing less than 50 percent extra node expansions for $\Lambda$ equal to $1.25$ times the Euclidean distance. The effect is almost independent of the page size.

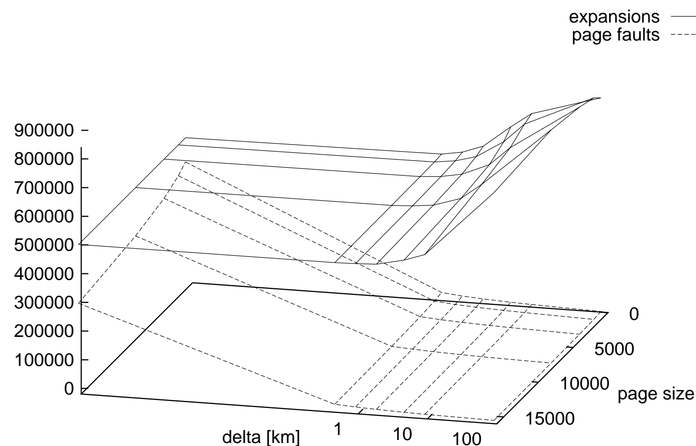


Figure 18.2: Number of page-faults and node expansions for varying page size and activeness bonus $\Delta$.

Unfortunately, the convincing decrease in page faults did not translate proportionally to execution time; the maximum reduction amounted to about 30 percent. We suspect that the reason is that we could not totally control the operating system's hardware paging still

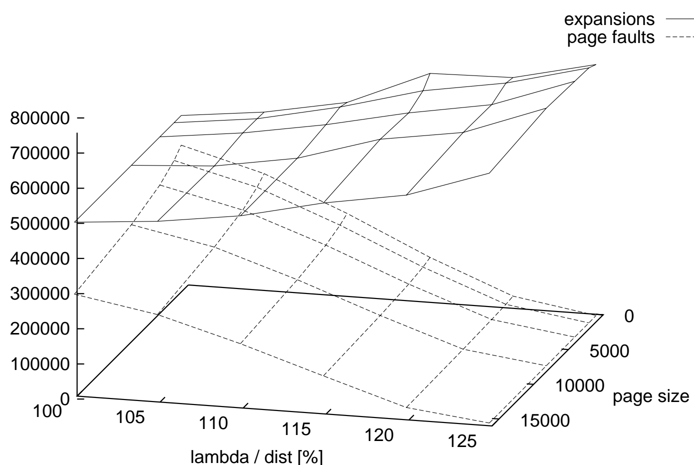


Figure 18.3: Number of page-faults and node expansions for varying page size and the ratio (in percent) of solution length approximation $\Lambda$ and the Euclidean distance *dist* between start and goal.

working besides and on top of our software paging technique. Hence, more inquiry into the platform-dependent implementation is still required.

We conclude that there is a trade-off between the growth of node expansions and the savings of page faults that has to be resolved by tuning the parameters to improve the overall efficiency for best performance.

## 18.5 Related Work

A couple of dynamic data structures have been proposed which take into account secondary memory structures. Major representatives are tree structured indices, such as B-Trees invented by Bayer and McCreight [23] and dynamic hashing variants [234], such as extendible hashing [120] and virtual hashing [240]. External sorting algorithms [210] are special-tailored for handling sequences on disk storage that do not fit into working memory. An extension for the LEDA [262] C++ library project to secondary storage systems, LEDA-SM for short, is being developed by Crauser and Mehlhorn at MPI/Saarbrücken.

One currently deeply investigated area in which the advantage of memory locality pays off is the breadth-first synthesis of binary decision diagrams (BDDs) [50]. The idea is to construct the diagram structure in a level-wise traversal [193]. Since there is a trade-off between low memory overhead and memory access locality, hybrid approaches based on context switches are currently being explored [351].

Since each page is explored independently, the algorithms easily lends itself to parallelization by allowing for more than one active page at a time. In fact, a commonly used method for duplicate pruning uses a hash function similar to $\phi$ defined above to associate with each node of the search space a distinct subspace with a dedicated processor. In [250], the notion of locality is important to reduce communication between processors

and it is implemented as the neighborhood on a hypercube.

There are some related approaches to the re-weighting technique used in our optimality proof. Searching negatively weighted graphs has been intensively studied in literature, cf [70]. An $O(|V||E|)$ algorithm for the single-source shortest path problem has been separately proposed by Bellman and Ford. The algorithm has been improved by Yen. The all-pair shortest path problem has been solved by Floyd based on a theorem of Warshall. It has been extended by Johnson for sparse and possibly negatively weighted graphs by re-weighting. All these algorithms do not apply to the scenario of implicitly given graphs with additional heuristic information.

## 18.6 Conclusion

We have presented an approach to relax the order of node expansions in traditional $A*$. Its admissibility is shown using a refined invariance condition based on Dijkstra's algorithm and re-weighted graphs. The reordering is used to make the search algorithm take into account memory locality for the price of an increased number of expansions. However, this is offset by the minimization of secondary memory access in a two-layered storage system, which is a major bottleneck for the traditional algorithm. To this end, the data structure *Heap-Of-Heaps* has been developed which partitions the underlying graph into pages; two heuristic threshold values discourage page switches and can be tuned for best performance. The count of page switches from an evaluation within a commercially available route planning system supports this view.

# Paper 19

# Route Planning and Map Inference with Global Positioning Traces

Stefan Edelkamp
Institut für Informatik
Am Flughafen 17
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

Stefan Schrödl
DaimlerChrysler Research and Technology
1510 Page Mill Road
Palo Alto, CA 94303
schroedl@rtna.daimlerchrysler.com

Accepted for publication.

**Abstract**

Navigation systems assist almost any kind of motion in the physical world including sailing, flying, hiking, driving and cycling. On the other hand, traces supplied by global positioning systems (GPS) can track actual time and absolute coordinates of the moving objects.

Consequently, this paper addresses efficient algorithms and data structures for the route planning problem based on GPS data; given a set of traces and a current location, infer a short(est) path to the destination.

The algorithm of Bentley and Ottmann is shown to transform geometric GPS information directly into a combinatorial weighted and directed graph structure, which in turn can be queried by applying classical and refined graph traversal algorithms like Dijkstras' single-source shortest path algorithm or A*.

For high-precision map inference especially in car navigation, algorithms for road segmentation, map matching and lane clustering are presented.

## 19.1 Introduction

Route planning is one of the most important application areas of computer science in general and graph search in particular. Current technology like hand-held computers, car navigation and GPS positioning systems ask for a suitable combination of mobile computing and course selection for moving objects.

In most cases, a possibly labeled weighted graph representation of all streets and crossings, called the *map*, is explicitly available. This contrasts other exploration problems like puzzle solving, theorem proving, or action planning, where the underlying problem graph is implicitly described by a set of rules.

Applying the standard solution of Dijkstra's algorithm for finding the single-source shortest path (SSSP) in weighted graphs from an initial node to a (set of) goal nodes faces several subtle problems inherent to route planning:

1. Most maps come on external storage devices and are by far larger than main memory capacity. This is especially true for on-board and hand-held computer systems.

2. Most available digital maps are expensive, since exhibiting and processing road information e.g. by surveying methods or by digitizing satellite images is very costly.

3. Maps are likely to be inaccurate and to contain systematic errors in the input sources or inference procedures.

4. It is costly to keep map information up-to-date, since road geometry continuously changes over time.

5. Maps only contain information on road classes and travel distances, which is often not sufficient to infer travel time. In rush hours or on bank holidays, the time needed for driving deviates significantly from the one assuming usual travel speed.

6. In some regions of the world digital maps are not available at all.

The paper is subdivided into two parts. In the first part, it addresses the process of map construction based on recorded data. In Section 19.2, we introduce some basic definitions. We present the *travel graph inference problem*, which turns out to be a derivate of the output sensitive sweep-line algorithm of Bentley and Ottmann. Subsequently, Section 19.3 describes an alternative statistical approach. In the second part, we provide solutions to accelerate SSSP computations for time or length optimal route planning in an existing accurate map based on Dijkstra's algorithm, namely A* with the Euclidean distance heuristic and refined implementation issues to deal with the problem of restricted main memory.

## 19.2 Travel Graph Construction

Low-end GPS data devices with accuracies of about 2-15 m and mobile data loggers (e.g. in form of palmtop devices) that store raw GPS data entries are nowadays easily accessible and widly distributed. To visualize data in addition to electronic road maps,

```
# latitude, longitude, date (yyyymmdd), time (hhmmss)

48.0131754,7.8336987,20020906,160241
48.0131737,7.8336991,20020906,160242
48.0131720,7.8336986,20020906,160243
48.0131707,7.8336984,20020906,160244
48.0131716,7.8336978,20020906,160245
48.0131713,7.8336975,20020906,160246
```

Figure 19.1: Small GPS trace.

recent software allows to include and calibrate maps from the Internet or other sources. Moreover, the adaption and visualization of topographical maps is no longer complicated, since high-quality maps and visualization frontends are provided at low price from organizations like *the Surveying Authorities of the States of the Federal Republic of Germany* with the TK50 CD series. Various 2D and 3D user interfaces with on-line and off-line tracking features assist the preparation and the reflection of trips.

In this section we consider the problem of generating a travel graph given a set of traces, that can be queried for shortest paths. For the sake of clarity, we assume that the received GPS data is accurate and that at each inferred crossing of traces, a vehicle can turn into the direction that another vehicle has taken.

With current technology of global positioning systems, the first assumption is almost fulfilled: on the low end, (differential) GPS yields an accuracy in the range of a few meters; high end positioning systems with integrated inertial systems can even achieve an accuracy in the range of centimeters.

The second assumption is at least feasible for hiking and biking in unknown terrain without bridges or tunnels. To avoid these complications especially for car navigation, we might distinguish valid from invalid crossings. Invalid crossing are ones with an intersection angle above a certain threshold and difference in velocity outside a certain interval. Fig. 19.1 provides a small example of a GPS trace that was collected on a bike on the campus of the computer science department in Freiburg.

### 19.2.1 Notation

We begin with some formal definitions. *Points* in the plane are elements of $\mathbb{R} \times \mathbb{R}$, and *line segments* are pairs of points. A *timed point* $p = (x, y, t)$ has global coordinates $x$ and $y$ and additional time stamp $t$, where $t \in \mathbb{R}$ is the absolute time to be decoded in year, month, day, hour, minute, second and fractions of a second. A *timed line segment* is a pair of timed points. A *trace* $T$ is a sequence of timed points $p_1 = (x_1, y_1, t_1), \ldots, p_n = (x_n, y_n, t_n)$ such that $t_i$, $1 \leq i \leq n$, is increasing. A *timed path* $P = s_1, \ldots, s_{n-1}$ is the associated sequence of timed line segments with $s_i = (p_i, p_{i+1})$, $1 \leq i < n$. The angle of consecutive line segements on a (timed) path and the velocity on timed line segments are immediate consequences of the above definitions.

The *trace graph* $G_T = (V, E, d, t)$ is a directed graph defined by $v \in V$ if its coordinates $(x_v, y_v)$ are mentioned in $T$, $e = (u, v) \in E$ if the coordinates of $u$ and $v$ correspond to two successive timed points $(x_u, y_u, t_u)$ and $(x_v, y_v, t_v)$ in $T$, $d(e) = ||u - v||_2$, and

$t(e) = t_v - t_u$, where $||u - v||_2$ denotes the Euclidean distance between (the coordinates of) $u$ and $v$.

The *travel graph* $G'_T = (V', E', d, t)$ is a slight modification of $G_T$ including its line segment intersections. More formally, let $s_i \cap s_j = r$ denote that $s_i$ and $s_j$ intersect in point $p$, and let $I = \{(r, i, j) \mid s_i \cap s_j = r\}$ be the set of all intersections, then $V' = V \cup \{r \mid (r, i, j) \in I\}$ and $E' = E \cup E_a \setminus E_d$, where $E_d = \{(s_i, s_j) \mid \exists r : (r, i, j) \in I\}$, and $E_a = \{(p, r), (r, q), (p', r), (r, q') \in V' \times V' \mid (r, i, j) \in I$ and $r = (s_i = (p, q) \cap s_j = (p', q'))\}$. Note that intersection points $r$ have no time stamp. Once more, the new cost values for $e = (u, v) \in E' \setminus E$ are determined by $d(e) = ||u - v||_2$, and by $t(e) = t(e')d(e)/d(e')$ with respect to the original edge $e' \in E_d$. The latter definition of time assumes a uniform speed on every line segment, which is plausible on sufficiently small line segments.

The *travel graph* $G'_D$ of a set $D$ of traces $T_1, \ldots T_l$ is the travel graph of the union graph of the respective trace graphs $G_{T_1}, \ldots, G_{T_k}$, Where the union graph $G = (V, E)$ of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$.

For the sake of simplicity, we assume that all crossings are in *general position*, so that not more than two line segments intersect in one point. This assumption is not a severe restriction, since all algorithms can be adapted to the more general case. We also might exclude matching endpoints from the computation, since we already know that two consecutive line segments intersect at the recorded data point. If a vehicle stops while the GPS recorder is running, zero-length sequences with strictly positive time delays are generated. Since zero-length segments cannot yield crossings, the problem of self loops might be dealt with ignoring these segments for travel graph generation and a re-inserting them afterwards to allow timed shortest path queries.

### 19.2.2 Algorithm of Bentley and Ottmann

The plane-sweep algorithm of Bentley and Ottmann [27] infers an undirected planar graph representation (the *arrangement*) of a set of segments in the plane and their intersections. The algorithm is one of the most innovative schemes both from a conceptual and from a algorithmical point of view.

From a conceptional point of view it combines the two research areas of *computational complexity* and *graph algorithms*. The basic principle of an imaginary *sweep-line* that stops on any interesting event is one of the most powerful technique in geometry e.g. to directly compute the Voronoi diagram on a set of $n$ points in optimal time $O(n \log n)$, and is a design paradigm for solving many combinatorial problems like the minimum and maximum in a set of values in the optimal number of comparisons, or the maximum sub-array sum in linear time with respect to the number of elements.

From an algorithmical point of view the algorithm is a perfect example of the application of balanced trees to reduce the complexity of an algorithm. It is also the first *output-sensitive algorithm*, since its time complexity $O((n + k) \log n)$ is measured in both the input and the output length, due to the fact that $n$ input segments may give rise to $k = O(n^2)$ intersections.

The core observation for route planning is that, given a set of traces $D$ in form of a sequence of segments, the algorithm can easily be adapted to compute the corresponding travel graph $G'_D$. In difference to the original algorithm devised for computational geometry problems, the generated graph structure has to be directed. The direction of each

edge $e$ as well as its distance $d(e)$ and travel time $t(e)$ is determined by the two end nodes of the segment. This includes intersections: the newly generated edges inherit direction, distance and time from the original end points.

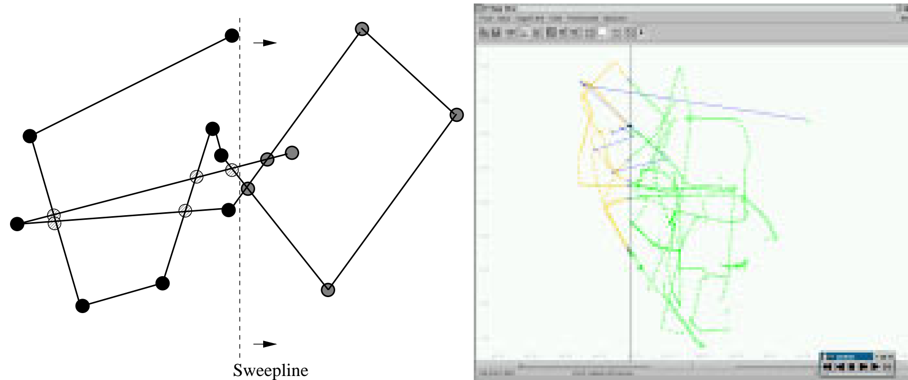


Figure 19.2: Visualization of the sweep-line algorithm of Bentley and Ottmann on a $i)$ tiny and $ii)$ small data set in the client-server visualization Java frontend VEGA.

In Fig. 19.2 we depicted a snapshot of the animated execution of the algorithm in the client-server visualization Java frontend VEGA [175] $i)$ on a line segment sample set and $ii)$ on an extended trail according to Fig. 19.1. The sweep-line proceeds from left to right, with the completed graph to its left.

The algorithm utilizes two data structures: the *event queue* and the *status structure*. In the event queue the active points are maintained, ordered with respect to their $x$-coordinate. In the status structure the active set of segments with respect to the sweep line is stored in $y$-ordering. At each intersection the ordering of segments in the status structure may change. Fortunately, the ordering of segments that participate in the intersections simply reverses, allowing fast updates in the data structure. After new neighboring segments are found, their intersections are computed and inserted into the event queue. The abstract data structure needed for implementation are a priority queue for the event queue and a search tree with neighboring information for the status data structure. Using a standard heap for the former and a balance tree for the latter implementation yields an $O((n+k)\log n)$ time algorithm.

The lower bound of the problem's complexity is $\Omega(n \log n + k)$ and the first step to improve time performance was $O(n \log^2 n / \log\log n + k)$ [55]. The first $O(n \log n + k)$ algorithm [56] used $O(n+k)$ storage. The $O(n \log n + k)$ algorithm with $O(n)$ space is due to Balaban [19].

For trace graphs the $O((n+k)\log n)$ implementation is sufficiently fast in practice. As a small example trace file consider $n = 2^{16} = 65,536$ segment end points with $k = 2^8 = 256$ intersections. Then $n^2 = 2^{32} = 4,294,967,296$, while $(n+k)\log n = (2^{16}+2^8)\cdot 16 = 1,052,672$ and $n \log n + k = 1,048,832$.

## 19.3 Statistical Map Inference

Even without map information, on-line routing information is still available, e.g. a driving assistance system could suggest *you are 2 meters off to the right of the best route*, or *you*

*have not followed the suggested route, I will recompute the shortest path from the new position*, or *turn left in about 100 meter in a resulting angle of about 60 degrees*.

Nevertheless, route planning in trace graphs may have some limitations in the presentation of the inferred route to a human, since abstract maps compact information and help to adapt positioning data to the real-world.

In this section, an alternative approach to travel graphs is presented. It concentrates on the map inference and adaptation problem for car navigation only, probably the most important application area for GPS routing. One rationale in this domain is the following. Even in one lane, we might have several traces that overlap, so that the number of line segment intersections $k$ can increase considerably. Take $m/2$ parallel traces on this lane that intersect another parallel $m/2$ traces on the lane a single lane in a small angle, then we expect up to $\Theta(m^2)$ intersections in the worst case.

We give an overview of a system that automatically generates digital road maps that are significantly more precise and contain descriptions of lane structure, including number of lanes and their locations, and also detailed intersection structure. Our approach is a statistical one: we combine 'lots of bad' GPS data from a fleet of vehicles, as opposed to 'few but highly accurate' data obtained from dedicated surveying vehicles operated by specially trained personnel. Automated processing can be much less expensive. The same is true for the price of DGPS systems; within the next few years, most new vehicles will likely have at least one DGPS receiver, and wireless technology is rapidly advancing to provide the communication infrastructure. The result will be more accurate, cheaper, up-to-date maps.

## 19.3.1 Steps in the Map Refinement Process

Currently commercially available digital maps are usually represented as graphs, where the nodes represent intersections and the edges are unbroken road *segments* that connect the intersections. Each segment has a unique identifier and additional associated attributes, such as a number of *shape points* roughly approximating its geometry, the road type (e.g., highway, on/off-ramp, city street, etc), speed information, etc. Generally, no information about the number of lanes is provided. The usual representation for a a two-way road is by means of a single segment. In the following, however, we depart from this convention and view segments as unidirectional links, essentially splitting those roads in two segments of opposite direction. This will facilitate the generation of the precise geometry.

The task of map refinement is simplified by decomposing it into a number of successive, dependent processing steps. Traces are divided into subsections that correspond to the road segments as described above, and the geometry of each individual segment is inferred separately. Each segment, in turn, comprises a subgraph structure capturing its lanes, which might include splits and merges. We assume that the lanes of a segment are mostly parallel. In contrast to commercial maps, we view an intersection as a structured region, rather than a point. These regions limit the segments at points where the traces diverge and consist of unconstrained trajectories connecting individual lanes in adjacent segments.

The overall map refinement approach can be outlined as follows.

1. Collect raw DGPS data (traces) from vehicles as they drive along the roads. Cur-

rently, commercially available DGPS receivers output positions (given as longitude/latitude/altitude coordinates with respect to a reference ellipsoid) at a regular frequency between 0.1 and 1 Hz.

Optionally, if available, measurements gathered for the purpose of electronic safety systems (anti-lock brakes or electronic stability program), such as wheel speeds and accelerometers, can be integrated into the positioning system through a *Kalman filter* [162]. In this case, the step 2 (filtering or smoothing) can be accomplished in the same procedure.

2. Filter and resample the traces to reduce the impact of DGPS noise and outliers. If, unlike in the case of the Kalman filter, no error estimates are available, some of the errors can be detected by additional indicators provided by the receiver, relating to satellite geometry and availability of the differential signal; others (e.g., so-called *multipath errors*) only from additional plausibility tests, e.g., maximum acceleration according to a vehicle model. Resampling is used to balance out the bias of traces recorded at high sampling rates or at low speed. Details of the preprocessing are beyond the scope of the current paper and can be found in a textbook such as [284].

3. Partition the raw traces into sequences of segments by *matching* them to an initial base map. This might be a commercial digital map, such as that of Navigation Technologies, Inc. [275]. Section 19.3.2 presents an alternative algorithm for inferring the network structure from scratch, from a set of traces alone.

   Since in our case we are not constrained to a real-time scenario, it is useful to consider the context of sample points when matching them to the base map, rather than one point at a time. We implemented a map matching module that is based on a modified best-first path search algorithm based on the Dijkstra-scheme [80], where the matching process compares the DGPS points to the map shape points and generates a cost that is a function of their positional distance and difference in heading. The output is a file which lists, for each trace, the traveled segment IDs, along with the starting time and duration on the segment, for the sequence with minimum total cost (a detailed description of map matching is beyond the scope of this paper).

4. For each segment, generate a *road centerline* capturing the accurate geometry that will serve as a reference line for the lanes, once they are found. Our spline fitting technique will be described in Section 19.3.3.

5. Within each segment, cluster the perpendicular offsets of sample points from the road centerline to identify *lane* number and locations (cf. Section 19.3.4).

### 19.3.2 Map Segmentation

In the first step of the map refinement process, traces are decomposed into a sequence of sections corresponding to road segments. To this end, an initial base map is needed for map matching. This can either be a commercially available map, such as that of Navigation Technologies, Inc. [275]; or, we can infer the connectivity through a spatial clustering algorithm, as will be described shortly.

These two approaches both have their respective advantages and disadvantages. The dependence on a commercial input map has the drawback that, due to its inaccuracies (Navigation Technologies advertises an accuracy of 15 meters), traces sometimes are incorrectly assigned to a nearby segment. In fact, we experienced this problem especially in the case of highway on-ramps, which can be close to the main lanes and have similar direction.

A further disadvantage is that roads missing in the input map cannot be learned at all. It is impossible to process regions if no previous map exists or the map is too coarse, thus omitting some roads.

On the other hand, using a commercial map as the initial baseline associates additional attributes with the segments, such as road classes, street names, posted speeds, house numbers, etc. Some of these could be inferred from traces by related algorithms on the basis of average speeds, lane numbers, etc. Pribe and Rogers [297] describe an approach to learning traffic controls from GPS traces. An approach to travel time prediction is presented in [156]. However, obviously not all of this information can be independently recovered. Moreover, with the commercial map used for segmentation, the refined map will be more compatible and comparable with applications based on existing databases.

### Road Segment Clustering

In this section, we outline an approach to *inferring* road segments from a set of traces simultaneously. Our algorithm can be regarded as a *spatial clustering procedure*. This class of algorithms is often applied to recognition problems in image processing (See e.g. [85] for an example of road finding in aerial images). In our case, the main questions to answer are to identify common segments used by several traces, and to locate the branching points (intersections). A procedure should be used that exploits the contiguity information and temporal order of the trace points in order to determine the connectivity graph. We divide it into three stages: cluster seed location, seed aggregation into segments, and segment intersection identification.

**Cluster Seed Location** *Cluster seed location* means finding a number of sample points on different traces belonging to the same road. Assume we have already identified a number of trace points belonging to the same cluster; from these, a mean values for position and heading is derived. In view of the later refinement step described in Sec. 19.3.3, we can view such a cluster center as one point of the *road centerline*.

Based on the assumption of lane parallelism, we measure the distance between traces by computing their intersection point with a line through the cluster center that runs perpendicular to the cluster heading; this is equivalent to finding those points on the traces whose projection onto the tangent of the cluster coincides with the cluster location, see Fig. 19.3.

Our similarity measure between a new candidate trace and an existing cluster is based both on its minimum distance to other member traces belonging to the cluster, computed as described above; and on the difference in heading. If both of these indicators are below suitable thresholds (call them $\theta$ and $\delta$, respectively) for two sample points, they are deemed to belong to the same road segment.

The maximum heading difference $\delta$ should be chosen to account for the accuracy of the data, such as to exclude sample points significantly above a high quantile of the error

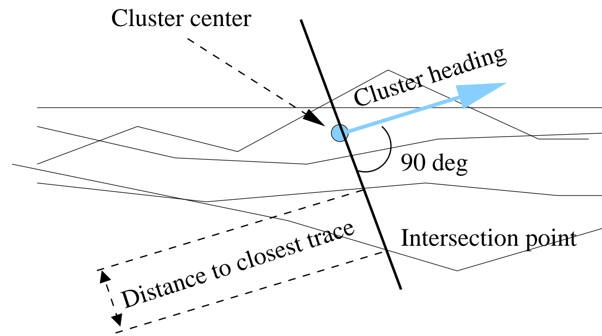


Figure 19.3: Distance between candidate trace and cluster seed

distribution. If such an estimate is unavailable, but a number of traces have already been found to belong to the cluster, the standard deviation of these members can give a clue. In general we found that the algorithm is not very sensitive to varation of $\delta$.

The choice of $\theta$ introduces a trade-off between two kinds of segmentation errors: if it is too small, wide lanes will be regarded as different roads; in the opposite case, nearby roads would be identified. In any case, the probablility that the GPS error exceeds the difference between the distance to the nearest road and the lane width is a lower bound for the segmentation error. A suitable value depends on the expected GPS errors, characteristics of the map (e.g., the relative frequencies of four-way intersections vs. freeway ramps), and also on the relative risks associated with both tyes of errors which are ultimately determined by the final application. As a conservative lower bound, $\theta$ should be at least larger than the maximum lane width, plus a tolerance (estimated standard deviation) for driving off the center of the lane, plus a considerable fraction of an estimated standard deviation of the GPS error. Empirically, we found the results with values in the range of 10–20 meters to be satisfying and sufficiently stable.

Using this similarity measure, the algorithm now proceeds in a fashion similar to the $k$-means algorithm [249]. First, we initialize the cluster with some random trace point. At each step, we add the closest point on any of the traces not already in the cluster, unless $\theta$ or $\delta$ is exceeded. Then, we recompute the average position and heading of the cluster center. Due to these changes, it can sometimes occur that trace points previously contained in the cluster do no longer satisfy the conditions for being on the same road; in this case they are removed. This process is repeated, until no more points can be added.

In this manner, we repeatedly generate cluster seeds at different locations, until each trace point has at least one of them within reach of a maximum distance threshold $d_{max}$. This threshold should be in an order of magnitude such that we ensure not to miss any intersection (say, e.g., 50 meters). A simple greedy strategy would follow each trace and add a new cluster seed at regular intervals of length $d_{max}$ when needed. An example section of traces, together with the generated cluster centers, are shown in Fig. 19.4.

**Segment Merging** The next step is to *merge* those ones of the previously obtained cluster centers that belong to the same road. Based on the connectivity of the traces, two such clusters $C_1$ and $C_2$ can be characterized in that (1) w.l.o.g. $C_1$ precedes $C_2$, i.e., all the traces belonging to $C_1$ subsequently pass through $C_2$, and (2) all the the traces belonging to $C_2$ originate from $C_1$. All possible adjacent clusters satisfying this condition are merged. A resulting maximum chain of clusters $C_1, C_2, \ldots, C_n$ is called a *segment*, and

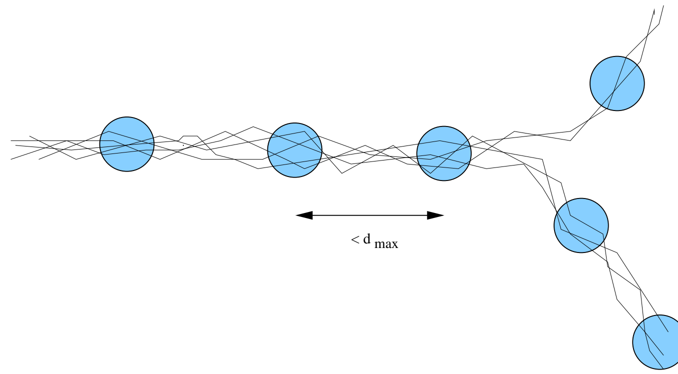


Figure 19.4: Example of traces with cluster seeds

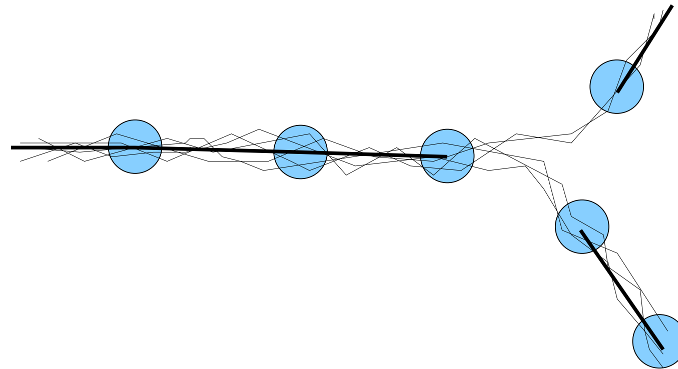

Figure 19.5: Merged cluster seeds result in segments

$C_1$ and $C_n$ are called the *boundary clusters* of the segment.

At the current segmentation stage of the map refinement process, only a crude geometric representation of the segment is sufficient; its precise shape will be derived later in the road centerline generation step (Sec. 19.3.3). Hence, as an approximation, adjacent cluster centers can either be joined by straight lines, polynomials, or one representative trace part (in our algorithms, we chose the latter possibility). In Fig. 19.5, the merged segments are connected with lines.

**Intersections** The only remaining problem is now to represent intersections. To capture the extent of an intersection more precisely, we first try to advance the boundary clusters in the direction of the split- or merge zone. This can be done by selecting a point from each member trace at the same (short) travel distance away from the respective sample point belonging to the cluster, and then again testing for contiguity as described above. We extend the segment iteratively in small increments, until the test fails.

The set of adjacent segments of an intersection is determined by (1) selecting all outgoing segments of one of the member boundary clusters; (2) collecting all incoming segments of the segments found in (1); and iterating these steps until completion. Each adjacent segment should be joined to each other segment for which connecting traces exist.

We utilize the concept of a *snake* borrowed from the domain of image processing, i.e., a contour model that is fit to (noisy) sample points. In our case, a simple star-shaped contour suffices, with the end points held fixed at the boundary cluster centers of the

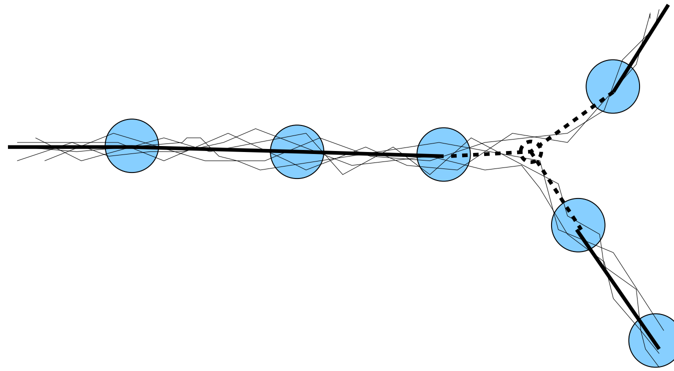

Figure 19.6: Traces, segments, and intersection contour model (dotted)

adjacent segments. Conceptually, each sample points exert an attracting force on it closest edge. Without any prior information on the shape of the intersection, we can define the 'energy' to be the sum of the squared distances between each sample point and the closest point on any of the edges, and then iteratively move the center point in an EM-style fashion in order to minimize this measure. The dotted lines in Fig. 19.6 correspond to the resulting snake for our example.

#### Dealing with Noisy Data

Gaps in the GPS receiver signal can be an error source for the road clustering algorithm. Due to obstructions, it is not unusual to find gaps in the data that span a minute. As a result, interpolation between distant points is not reliable.

As mentioned above, checking for parallel traces crucially depends on *heading* information. For certain types of positioning systems used to collect the data, the heading might have been determined from the direction of differences between successive sample points. In this case, individual outliers, and also lower speeds, can lead to even larger errors in direction.

Therefore, in the first stage of our segmentation inference algorithm, filtering is performed by disregarding trace segments in cluster seeds that have a gap within a distance of $d_{max}$, or fall outside a 95 percent interval in the heading or lateral offset from the cluster center. Cluster centers are recomputed only from the remaining traces, and only they contribute to the subsequent steps of merging and intersection location with adjacent segments.

Another issue concerns the start and end parts of traces. Considering them in the map segmentation could introduce segment boundaries at each parking lot entrance. To avoid a too detailed breakup, we have to disregard initial and final trace sections. Different heuristics can be used; currently we apply a combined minimum trip length/speed threshold.

### 19.3.3 Road Centerline Generation

We now turn our attention to the refinement of individual segments. The *road centerline* is a geometric construct whose purpose is to capture the road geometry. The road centerline can be thought of as a weighted average trace, hence subject to the relative lane

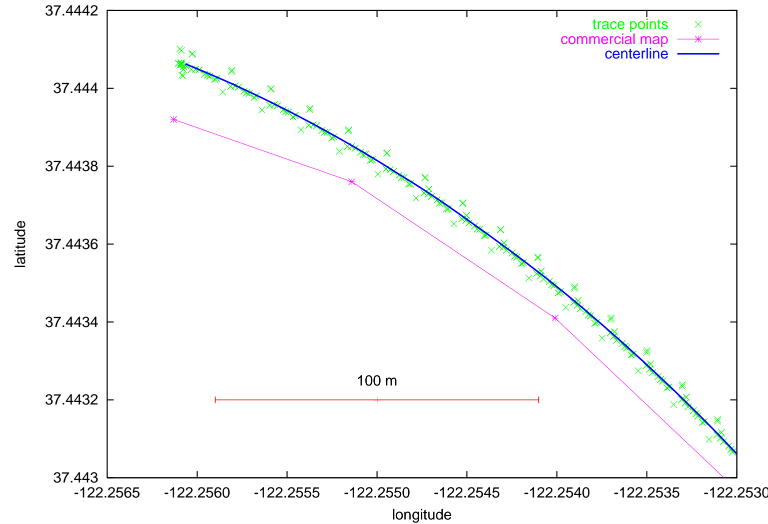


Figure 19.7: Segment part: NavTech map (bottom), trace points (crosses), and computed centerline

occupancies, and not necessarily a trajectory any single vehicle would ever follow. We assume, however, the lanes to be parallel to the road centerline, but at a (constant) perpendicular offset. For the subsequent lane clustering, the road centerline helps to cancel out the effects of curved roads.

For illustration, Fig. 19.7 shows a section of a segment in our test area. The indicated sample points stem from different traces. Clearly, by comparison, the shape points of the respective NavTech segment exhibit a systematic error. The centerline derived from the sample points is also shown.

It is useful to represent our curves in *parametric form*, i.e., as a vector of coordinate variables $C(u) = (x, y, z)(u)$ which is a function of an independent parameter $u$, for $0 \leq u \leq 1$. The centerline is generated from a set of sample points using a *weighted least squares fit*. More precisely, assume that $Q_0, \ldots, Q_{m-1}$ are the $m$ data points given, $w_0, \ldots, w_{m-1}$, are associated weights (dependent on an error estimate), and $\bar{u}_0, \ldots, \bar{u}_{m-1'}$ their respective parameter values. The task can be formulated as finding a parametric curve $C(u)$ from a class of functions $\mathcal{S}$ such that the $Q_k$ are approximated in the weighted least square sense, i.e.

$$s := \sum_{k=0}^{m-1} w_k \cdot \|Q_k - C(\bar{u}_k)\|^2$$

in a minimum with respect to $\mathcal{S}$, where $\|\cdot\|$ denotes the usual Euclidean distance (2-norm). Optionally, in order to guarantee continuity across segments, the algorithm can easily be generalized to take into account derivatives; if heading information is available, we can use coordinate transformation to arrive at the desired derivative vectors.

The class $\mathcal{S}$ of approximating functions is composed of rational *B-Splines*, i.e., piecewise defined polynomials with continuity conditions at the joining knots (for details, see [293, 321]). For the requirement of continuous curvature, the degree of the polynomial has to be at least three.

If each sample point is marked with an estimate of the measurement error (standard deviation), which is usually available from the receiver or a Kalman filter, then we can

use its inverse to weight the point, since we want more accurate points to contribute more to the overall shape.

The least squares procedure [293] expects the *number of control points* $n$ as input, the choice of which turns out to be critical. The control points define the shape of the spline, while not necessarily lying on the spline themselves. We will return to the issue of selecting an adequate number of control points in Section 19.3.3.

### Choice of Parameter Values for Trace Points

For each sample point $Q_k$, a parameter value $\bar{u}_k$ has to be chosen. This parameter vector affects the shape and parameterization of the spline. If we were given a single trace as input, we could apply the widely used *chord length* method as follows. Let $d$ be the total chord length $d = \sum_{k=1}^{m-1} |Q_k - Q_{k-1}|$. Then set $\bar{u}_0 = 0$, $\bar{u}_{m-1} = 1$, and $\bar{u}_k = \bar{u}_{k-1} + \frac{|Q_k - Q_{k-1}|}{d}$ for $k = 1, \ldots, m-2$. This gives a good parameterization, in the sense that it approximates a *uniform* parameterization proportional to the arc length.

For a set of $k$ distinct traces, we have to impose a common ordering on the combined set of points. To this end, we utilize an initial rough approximation, e.g., the polyline of shape points from the original NavTech map segment $s$; if no such map segment is available, one of the traces can serve as a rough baseline for projection. Each sample point $Q_k$ is *projected* onto $s$, by finding the closest interpolated point on $s$ and choosing $\bar{u}_k$ to be the chord length (cumulative length along this segment) up to the projected point, divided by the overall length of $s$. It is easy to see that for the special case of a single trace identical to $s$, this procedure coincides with the chord length method.

### Choice of the Number of Control Points

The number of control points $n$ is crucial in the calculation of the centerline; for a cubic spline, it can be chosen freely in the valid range $[4, m-1]$. Fig. 19.8 shows the centerline for one segment, computed with three different parameters $n$.

Note that a low number of control points may not capture the shape of the centerline sufficiently well ($n = 4$); on the other hand, too many degrees of freedom causes the result to “fit the error”. Observe how the spacing of sample points influences the spline for the case $n = 20$.

From the latter observation, we can derive an upper bound on the number of control points: it should not exceed the average number of sample points per trace, multiplied by a small factor, e.g., $2 * m/k$.

While the appropriate number of control points can be easily estimated by human inspection, its formalization is not trivial. We empirically found that two measures are useful in the evaluation.

The first one is related to the *goodness of fit*. Averaging the absolute offsets of the sample points from the spline is a feasible approach for single-lane roads, but otherwise depends on the number and relative occupancies of lanes, and we do not expect this offset to be zero even in the ideal case. Intuitively, the centerline is supposed to stay roughly in the middle between all traces; i.e., if we project all sample points on the centerline, and imagine a fixed-length window moving along the centerline, then the average offset of all sample points whose projections fall into this window should be near to zero. Thus, we define the *approximation error* $\epsilon_{\text{fit}}$ as the average of these offsets over all windows.

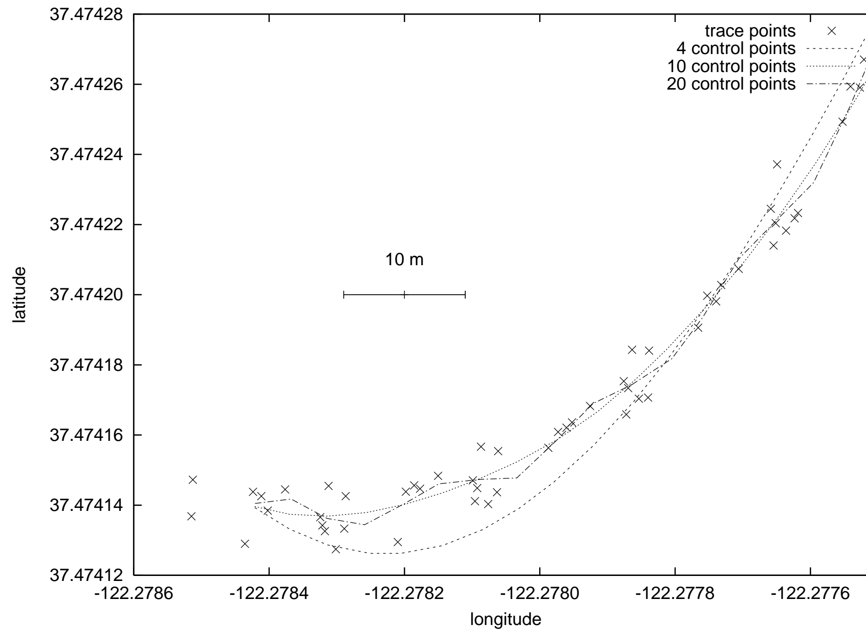


Figure 19.8: Trace points and centerlines computed with varying number of control points

The second measure checks for overfitting. As illustrated in Fig. 19.8, using a large number of control points renders the centerline "wiggly", i.e., tends to increase the *curvature* and makes it change direction frequently. However, according to construction guidelines, roads are designed to be piecewise segments of either straight lines or circles, with clothoids between as transitions. These geometric concepts constrain the curvature to be *piecewise linear*. As a consequence, the second derivative of the curvature is supposed to be zero nearly everywhere, with the exception of the segment boundaries where it might be singular. Thus, we evaluate the curvature of the spline at constant intervals and numerically calculate the second derivative. The average of these values is the *curvature error* $\epsilon_{\text{curv}}$.

Fig. 19.9 plots the respective values of $\epsilon_{\text{fit}}$ and $\epsilon_{\text{curv}}$ for the case of Fig. 19.8 as a function of the number of control points. There is a tradeoff between $\epsilon_{\text{fit}}$ and $\epsilon_{\text{curv}}$; while the former tends to decrease rapidly, the latter increases. However, both values are not completely monotonic.

Searching the space of possible solutions exhaustively can be expensive, since a complete spline fit has to be calculated in each step. To save computation time, the current approach heuristically picks the largest valid number of control points for which $\epsilon_{\text{curv}}$ lies below an acceptable threshold.

### 19.3.4 Lane Finding

After computing the approximate geometric shape of a road in the form of the road centerline, the aim of the next processing step is to infer the number and positions of its *lanes*. The task is simplified by canceling out road curvature by the following transformation. Each trace point $P$ is *projected* onto the centerline for the segment, i.e., its nearest interpolated point $P'$ on the map is determined. Again, the arc length from the first centerline

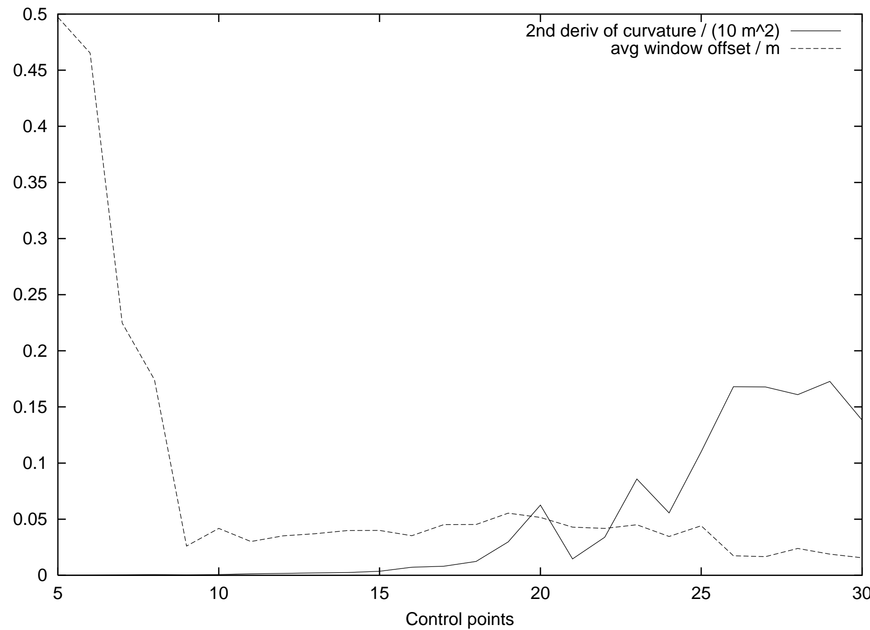


Figure 19.9: Error Measures $\epsilon_{\text{fit}}$ and $\epsilon_{\text{curv}}$ vs Number of Control Points

point up to $P'$ is the *distance along the segment*; the distance between $P$ and $P'$ is referred to as its *offset*. An example of the transformed data is shown in Fig. 19.10.

Intuitively, clustering means assigning $n$ data points in a $d$-dimensional space to $k$ clusters such that some distance measure within a cluster (i.e., either between pairs of data belonging to the same cluster, or to a cluster center) is minimized (and is maximized between different clusters). For the problem of lane finding, we are considering points in a plane representing the flattened face of the earth, so the Euclidean distance measure is appropriate.

Since clustering in high-dimensional spaces is computationally expensive, methods like the *$k$-means algorithm* use a hill-climbing approach to find a (local) minimum solution. Initially, $k$ cluster centers are selected, and two phases are iteratively carried out until cluster assignment converges. The first phase assigns all points to their nearest cluster center. The second phase recomputes the cluster center based on the respective constituent points (e.g., by averaging) [249].

**Segments with Parallel Lanes** If we make the assumption that lanes are parallel over the entire segment, the clustering is essentially one-dimensional, taking only into account the offset from the road centerline. In our previous approach [308], a hierarchical agglomerative clustering algorithm (*agglom*) was used that terminated when the two closest clusters were more than a given distance apart (which represented the maximum width of a lane). However, this algorithm requires $O(n^3)$ computation time. More recently, we have found that it is possible to explicitly compute the *optimal solution* in $O(k \cdot n^2)$ time and $O(n)$ space using a dynamic programming approach, where $n$ denotes the number of sample points, and $k$ denotes the maximum number of clusters [321].

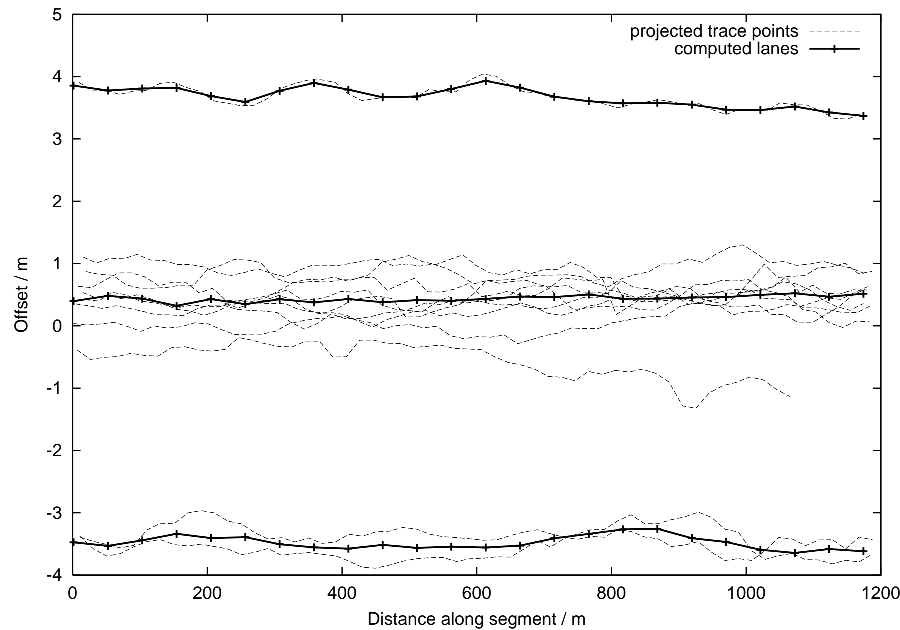


Figure 19.10: Traces projected on centerline (dashed), computed lane centers (solid)

### Segments with Lane Splits and Merges

We have previously assumed the ideal case of a constant number of lanes at constant offsets from the road centerline along the whole segment. However, lane widths may gradually change; in fact, they usually get wider near an intersection. Moreover, new lanes can start at any point within a segment (e.g., turn lanes), and lanes can merge (e.g., on-ramps). Subsequently, we present two algorithms that can accommodate the additional complexity introduced by lane merges and splits.

**One-dimensional Windowing with Matching** One solution to this problem is to augment the one-dimensional algorithm with a windowing approach. We divide the segment into successive windows with centers at constant intervals along the segment. To minimize discontinuities, we use a Gaussian convolution to generate the windows. Each window is clustered separately as described above, i.e., the clustering essentially remains one-dimensional. If the number of lanes in two adjacent windows remains the same, we can associate them in the order given. For lane splits and merges, however, lanes in adjacent windows need to be *matched*.

To match lanes across window boundaries, we consider each trace individually (the information as to which trace each data point belongs to has not been used in the centerline generation, nor the lane clustering). Following the trajectory of a given trace through successive windows, we can classify its points according to the computed lanes. Accumulating these counts over all traces yields a matrix of *transition frequencies* for any pair of lanes from two successive windows. Each lane in a window is matched to that lane in the next window with maximum transition frequency.

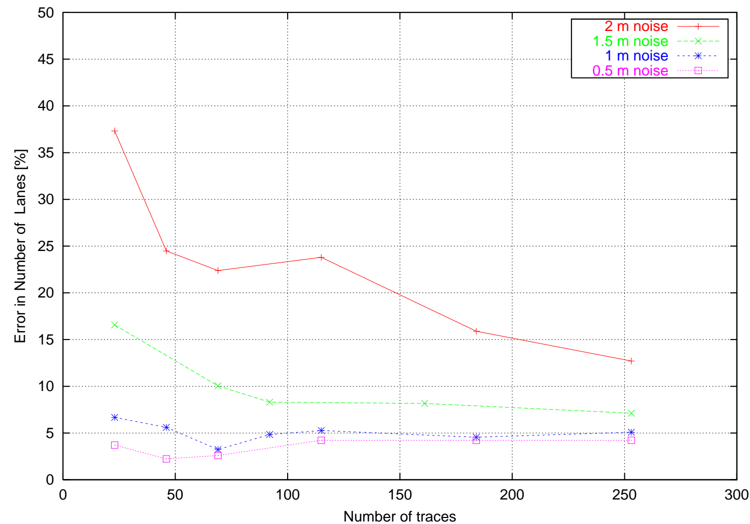


Figure 19.11: Error in determining the number of lane clusters

### 19.3.5 Experimental Results

We have now completed our account of the individual steps in the map refinement process. In this section, we report on experiments we ran in order to evaluate the learning rate of our map refinement process. Our test area in Palo Alto, CA, covered 66 segments with a combined length of approximately 20 km of urban and freeway roads of up to four lanes, with an average of 2.44 lanes.

One principal problem we faced was the availability of a ground truth map with lane-level accuracy for comparison. Development of algorithms to integrate vision-based lane tracker information is currently under way. Unfortunatly, however, these systems have errors on their own and therefore cannot be used to gauge the accuracy of the pure position-based approach described in this article. Therefore, we reverted to the following procedure. We used a high-end real-time kinematic carrier phase DGPS system to generate a base map with few traces [347]. According to the announced accuracy of the system of about 5 cm, and after visual inspection of the map, we decided to define the obtained map as our baseline. Specifically, the input consisted of 23 traces at different sampling rates between 0.25 and 1Hz.

Subsequently, we artificially created more traces of lower quality by adding varying amounts of gaussian noise to each individual sample position ($\sigma = 0.5 \ldots 2$ m) of copies of the original traces. For each combination of error level and training size, we generated a map and evaluated its accuracy.

Fig. 19.11 shows the resulting error in the *number of lanes*, i.e., the proportion of instances where the number of lanes in the learned map differs from the number of lanes in the base line map at the same position. Obviously, for a noise level in the range of more than half a lane width, it becomes increasingly difficult to distinguish different clusters of road centerline offsets due to overlap. Therefore, the accuracy of the map for the input noise level of $\sigma = 2$ m is significantly higher than that of the lower ones. However, based on the total spread of the traces, the number of lanes can still be estimated. For $n = 253$ traces, their error is below 10 percent for all of them. These remaining differences arise mainly from splits and merges, where in absence of the knowledge of lane markings it is

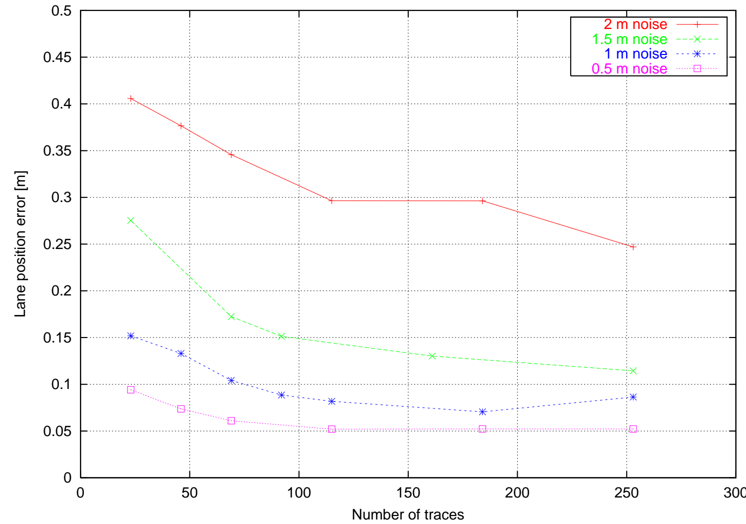


Figure 19.12: Average error in lane offsets

hard to determine the exact branch points, whose position can heavily depend on single lane changes in traces.

Fig. 19.12 plots the mean absolute difference of the offsets of corresponding lanes between the learned map and the base map, as a function of the training size (number of traces). Again, the case $\sigma = 2$ m needs significantly more training data to converge. For $\sigma <= 1.5$ m, the lane offset error decreases rapidly; it is smaller than 15 centimeters after $n = 92$ traces, and thus in the range of the base map accuracy.

## 19.4 Searching the Map Graph

Let us now turn our attention to the map usage in on-line routing applications. In some sense, both maps and travel graphs can be viewed as embeddings of weighted general graphs. Optimal paths can be searched with respect to accumulated shortest time $t$ or distance $d$ or any combination of them. We might assume a linear combination for a total weight function $w(u, v) = \lambda \cdot t(u, v) + (1 - \lambda) \cdot d(u, v)$ with parameter $\lambda \in \mathbb{R}$ and $0 \leq \lambda \leq 1$.

### 19.4.1 Algorithm of Dijkstra

Given a weighted graph $G = (V, E, w)$, $|V| = n$, $|E| = e$, the shortest path between two nodes can be efficiently computed by Dijkstra's single source shortest path (SSSP) algorithm [80].

Table 19.1 shows a implementation of Dijkstra's algorithm for implicitly given graphs that maintains a visited list *Closed* in form of a hash table and a priority queue of the nodes to be expanded, ordered with respect to increasing merits $f$.

The run time of Dijkstra's algorithm depends on the priority queue data structure. The original implementation of Dijkstra runs in $O(n^2)$, the standard textbook algorithm in $O((n + e) \log n)$ [70], and utilizing Fibonacci-heaps we get $O(e + n \log n)$ [134]. If the weights are small, buckets are preferable. In a Dial the $i$-th bucket contains all

| **Dijkstra:** | **A*** |
|---|---|
| *Open* $\leftarrow \{(s,0)\}$ | *Open* $\leftarrow \{(s,h(s))\}$ |
| *Closed* $\leftarrow \{\}$ | |
| **while** (*Open* $\neq \emptyset$) | |
| $u \leftarrow$ *Deletemin*(*Open*) | |
| *Insert*(*Closed*,$u$) | |
| **if** $(goal(u))$ **return** $u$ | |
| **for all** $v$ **in** $\Gamma(u)$ | |
| $f'(v) \leftarrow f(u) + w(u,v)$ | $+h(v) - h(u)$ |
| **if** (*Search*(*Open*, $v$)) | |
| **if** $(f'(v) < f(v))$ | |
| *DecreaseKey*(*Open*($v, f'(v)$)) | |
| **else if not** (*Search*(*Closed*, $v$)) | |
| *Insert*(*Open*, $(v, f'(v))$) | |

Table 19.1: Implementation of Dijkstra's SSSP algorithm vs. A*.

elements with a $f$-value equal to $i$ [79]. Dials yields $O(e + n \cdot C)$ time for SSSP, with $C \leftarrow max_{(u,v) \in E}\{w(u,v)\}$. Two-Level Buckets have top level and bottom level of length $\lceil\sqrt{C+1}\rceil + 1$, yielding the run time $O(e + n\sqrt{C})$. An implementation with *radix heaps* uses buckets of sizes $1, 1, 2, 4, 8, \ldots$ and imply $O(e + n \log C)$ run time, two-level heap improve the bound to $O(e + n \log C / \log\log C)$ and a hybrid with Fibonacci heaps yields $O(e + n\sqrt{\log C})$ [5]. This algorithm is almost linear in practice, since when assuming 32 bit integers we have $\lceil\sqrt{\log C}\rceil \leq 6$. The currently best result are *component trees* with $O(n + e)$ time for undirected SSSP on a random access machine of word length $w$ with integer edge weights in $[0..2^w - 1]$ [340]. However, the algorithm is quite involved and likely not to be practical.

### 19.4.2 Planar Graphs

Travel graphs have many additional features. First of all, the number of edges is likely to be small. In the trail graph the number of edges equals the number of nodes minus 1, and for $l$ trails $T_1, \ldots, T_l$ we have $|T_1|, \ldots, |T_l| - l$ edges in total. By introducing $k$ intersections the number of edges increases by $2k$ only. Even if intersections coincide travel graphs are still planar, and by Eulers formula the number of edges is bounded by at most three times the number of nodes. Recall, that for the case of planar graphs linear time algorithms base on graph separators and directly lead to network flow algorithms of the same complexity [208].

If some intersections were rejected by the algorithms to allow non-intersecting crossing like bridges and tunnels, the graph would loose some of its graph theoretical properties. In difference to general graphs, however, we can mark omitted crossings to improve run time and storage overhead.

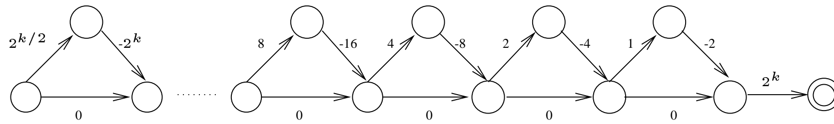


Figure 19.13: A graph with an exponential number of re-openings

### 19.4.3 Frontier Search

Frontier search [222] contributes to the observation that the newly generated nodes in any graph search algorithm form a connected horizon to the set of expanded nodes, which is omitted to save memory.

The technique refers to Hirschberg's linear space divide-and-conquer algorithm for computing maximal common sequences [176]. In other words, frontier search reduces a $(d+1)$-dimensional memorization problem into a $d$-dimensional one. It divides into three phases. In the first phase, a goal $t$ with optimal cost $f^*$ is searched. In the second phase the search is re-invoked with bound $f^*/2$; and by maintaining shortest paths to the resulting fringe the intermediate state $i$ from $s$ to $t$ is detected. In the last phase the algorithm is recursively called for the two subproblems from $s$ to $i$, and from $i$ to $t$.

### 19.4.4 Heuristic Search

Heuristic search includes an additional node evaluation function $h$ into the search. The estimate $h$, also called heuristic, approximates the shortest path distance from the current node to one of the goal nodes. A heuristic is *admissible* if it provides a lower bound to the shortest path distance and it is *consistent*, if $w(u,v) + h(v) - h(u) \geq 0$. Consistent estimates are admissible.

Table 19.1 also shows the small changes in the implementation of A* for consistent estimates with respect to Dijkstra's SSSP algorithm. In the priority queue *Open* of generated and not expanded nodes, the $f$-values are tentative, while in set *Closed* the $f$-values are settled. On every path from to the initial state to a goal node the accumulated heuristic values telescope, and if any goal node has estimate zero, the $f$ values of each encountered goal node in both algorithms are the same. Since in Dijkstra's SSSP algorithm the $f$-value of all expanded nodes match their graph theoretical shortest path value we conclude that for consistent estimates, A* is complete and optimal.

Optimal solving the SSSP problem for admissible estimates and negative values of $w(u,v) + h(v) - h(u)$ leads to re-openings of nodes: already expanded nodes in *Closed* are pushed back into the search frontier *Open*. If we consider $w(u,v) + h(v) - h(u)$ as the new edge costs, Fig. 19.13 gives an example for such a re-weighted graphs that leads to exponentially many re-openings. The second last node is re-opened for every path with weight $\{1, 2, \ldots, 2^k - 1\}$. Recall that if $h$ is consistent, no reopening will be necessary at all.

In route planning the Euclidean distance of two nodes is a heuristic estimate defined as $h(u) = \min_{g \in G} ||g - u||_2$ for the set of goal nodes $G$ is both admissible and consistent Admissibility is granted, since no path on any road map can be shorter than the flight distance, while consistency follows from the triangle inequality for shortest path. For edge $e = (u,v)$ we have $\min_{g \in G} ||g - v||_2 \leq \min_{g \in G} ||g - u||_2 + w(u,v)$ . Since nodes closer to the goal are more attractive, A* is likely to find the optimum faster. Another benefit

from this point of view is that all above advanced data structures for node maintenance in the priority as well as space saving strategies like frontier search can be applied to A*.

## 19.5 Related Work

Routing schemes often run on external maps and external maps call for refined memory maintenance. Recall that external algorithms are ranked according to *sorting complexity* $O(sort(n))$, i.e., the number of external block accesses (I/Os) necessary to sort $n$ numbers, and according to *scanning complexity* $O(scan(n))$, i.e., the number of I/Os to read $N$ numbers. The usual assumption is that $N$ is much larger than $B$, the block size. Scanning complexity equals $O(n/B)$ in a single disk model. On planar graphs, SSSP runs in $O(\mathit{sort}(n))$ I/Os, where $n$ is the number of vertices. As for the internal case the algorithms apply graph separation techniques [339]. For general BFS at most $O(\sqrt{n \cdot \mathit{scan}(n+e)} + \mathit{sort}(n+e))$ I/Os [261] are needed, where $e$ is the number of edges in the graph. Currently there is no $o(n)$ algorithm for external SSSP. On the other hand, $O(n)$ I/Os are by far too much in route planning practice.

Fortunately, one can utilize the spatial structure of a map to guide the secondary mapping strategy with respect to the graph embedding. The work of [115] provides the new search algorithm and suitable data structures in order to minimize page faults by a local reordering of the sequence of expansions. Algorithm *Localized A** introduces an operation *deleteSome* instead of strict *deleteMin* into the A* algorithm. Nodes corresponding to an active page are preferred. When maintaining an bound $\alpha$ on obtained solution lengths until *Open* becomes empty the algorithm can be shown to be complete and optimal. The loop invariant is that there is always a node on the optimal solution path with correctly estimated accumulated cost. The authors prove the correctness and completeness of the approach and evaluate it in a real-world scenario of searching a large road map in a commercial route planning system.

In many fields of application, shortest path finding problems in very large graphs arise. Scenarios where large numbers of on-line queries for shortest paths have to be processed in real-time appear for example in traffic information systems. In such systems, the techniques considered to speed up the shortest path computation are usually based on pre-computed information. One approach proposed often in this context is a space reduction, where pre-computed shortest paths are replaced by single edges with weight equal to the length of the corresponding shortest path. The work of [323] gives a first systematic experimental study of such a space reduction approach. The authors introduce the concept of multi-level graph decomposition. For one specific application scenario from the field of timetable information in public transport, the work gives a detailed analysis and experimental evaluation of shortest path computations based on multi-level graph decomposition.

In the scenario of a central information server in the realm of public railroad transport on wide area networks a system has to process a large number of on-line queries for optimal travel connections in real time. The pilot study of [322] focuses on travel time as the only optimization criterion, in which various speed-up techniques for Dijkstra's algorithm were analyzed empirically.

Speed-up techniques that exploit given node coordinates have proven useful for shortest-path computations in transportation networks and geographic information sys-

tems. To facilitate the use of such techniques when coordinates are missing from some, or even all, of the nodes in a network [44] generate artificial coordinates using methods from graph drawing. Experiments on a large set of train timetables indicate that the speed-up achieved with coordinates from network drawings is close to that achieved with the actual coordinates.

## 19.6 Conclusions

We have seen a large spectrum of efficient algorithms to tackle different aspects of the route planning problem based on a given set of global positioning traces.

For trail graph inference the algorithm of Bentley and Ottmann has been modified and shown to be almost as efficient as the fastest shortest path algorithms. Even though this solves the basic route planning problem, different enhanced aspects are still open. We indicate low memory consumption, localized internal computation, and fast on-line performance as the most challenging ones.

Map inference and map matching up to lane accuracy suite better as a human-computer interface, but the algorithmic questions include many statistical operations and are non-trivial for perfect control. On the other hand, map inference based on GPS information saves much money especially to structure unknown and continuously changing terrains.

Low-end devices will improve GPS accuracy especially by using additional inertia information of the moving object, supplied e.g. by a tachometer, an altimeter, or a compass. For (3D) map generation and navigation other sensor data like sonar and laser (scans) can be combined with GPS e.g. for outdoor navigation of autonomous robots and in order to close uncaught loops.

The controversery if the GPS routing problem is more a geometrical one (in which case the algorithm of Bentley/Ottmann applies) or a statistical one (in which clustering algorithms are needed) is still open. At the moment we expect statistical methods to yield better and faster results due to their data reduction and refinement aspects and we expect that a geometrical approach will not suffice to appropriately deal with a large and especially noisy data set. We have already seen over-fitting anomalies in the statistic analyses. Nevertheless, a lot more research is needed to clarify the the quest of a proper static analysis of GPS data, which in turn will have a large impact in the design and efficiency of the search algorithms.

We expect that in the near future, the combination of positioning and precision map technology will give rise to a range of new vehicle safety and convenience applications, ranging from warning, advice, up to automated control.

# Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining associating rules between sets of items in large databases. *ACM SIGMOD*, pages 207–216, 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] A. V. Aho, J. E. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, 1974.

[5] R. K. Ahuja, K. Mehlhorn, J. B. Orbin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, pages 213–223, 1990.

[6] J. F. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Franscico, CA, 1990.

[7] V. Allis. A knowledge-based approach to connect-four. The game is solved: White wins. Master's thesis, Vrije Univeriteit, The Netherlands, 1988.

[8] V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Department of Computer Science, Limburg, Maastrich, 1994.

[9] V. Allis, H. J. van den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Artificial Intelligence*, 66:91–124, 1993.

[10] R. Alur, R. Brayton, T. Henzinger, S. Qaderer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 340–351. Springer, 1997.

[11] D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981.

[12] D. Angluin. Inferrence of reversible languages. *Journal of the Association of Computing Machinery*, 29:741–765, 1982.

[13] V. V. Anshelevich. The game of Hex: An automatic theorem proving approach to game programming. In *National Conference on Artificial Intelligence (AAAI)*, pages 189–194, 2000.

[14] L. Arge. The I/O - complexity of ordered binary decision diagram manipulation. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 82–91, 1995.

[15] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, 1996.

[16] F. Bacchus and M. Ady. Planning with resources and concurrency: A forward chaning approach. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 417–424, 2001.

[17] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[18] C. Bäckström. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research*, 9:99–137, 1998.

[19] I. J. Balaban. An optimal algorithm for finding segment intersection. In *ACM Symposium on Computational Geometry*, pages 339–364, 1995.

[20] M. Baldamus, K. Schneider, M. Wenz, and R. Ziller. Can american checkers be solved by means of symbolic model checking? *Electronic Notes in Theoretical Computer Science*, 43, 2002.

[21] B. Barras, S. Boutin, C. Cornes, J. Courant, J. C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, 1997.

[22] D. Basin and H. Ganzinger. Automated complexity analysis based on ordered resolution. *Journal of the ACM*, 48(1):70–109, 2001.

[23] R. Bayer and E. McCreight. Organization and maintenanace of large ordered indexes. *Acta Informatica*, 13(7):427–436, 1970.

[24] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Petterson, and J. Romijn. Guiding and cost-optimality in UPPAAL. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 66–74, 2001.

[25] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Petterson, J. Romijn, and F. W. Vaandrager. Efficient guiding towards cost-optimality in uppaal. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science. Springer, 2001.

[26] L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:18–101, 1966.

[27] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Transactions on Computing*, 28:643–647, 1979.

[28] B. Bérard, A. F. M. Bidoit, F. Laroussine, A. Petit, L. Petrucci, P. Schoenebelen, and P. McKenzie. *Systems and Software Verification*. Springer, 2001.

[29] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning ways (book)*. Academic Press, vol.I and II, 850 pages, 1982.

[30] P. Bertoli and A. Cimatti. Improving heuristics for planning as search in belief space. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[31] P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search symbolic model checking = efficient conformant planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 467–472, 2001.

[32] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 473–478, 2001.

[33] J. Bevmyr. *Data Parallel Implementation of Prolog*. PhD thesis, Uppsala University, 1996.

[34] A. Biere. *Effiziente $\mu$-Kalkül-Prüfung mit Binären Entscheidungsdiagrammen*. PhD thesis, University of Karlsruhe, 1997.

[35] A. Biere. $\mu$cke - efficient $\mu$-calculus model checking. In *Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 468–471. Springer, 1997.

[36] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer, 1999.

[37] A. W. Biermann. The inference of regular lisp programs from examples. *IEEE Trans. on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

[38] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Conference on Design Automation (DAC)*, pages 29–34, 2000.

[39] A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1636–1642, 1995.

[40] B. Bonet and H. Geffner. Planning with incomplete information as heurstic search in belief space. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 52–61, 2000.

[41] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 2001.

[42] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *National Conference on Artificial Intelligence (AAAI)*, pages 714–719, 1997.

[43] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.

[44] U. Brandes, F. Schulz, D. Wagner, and T. Willhalm. Travel planning with self-made maps. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2001.

[45] J. H. Breasted. *The Edwin Smith Surgical Papyrus*, volume 1-2. The Oriental Institute of the University of Chicago, 1930 (Reissued 1991). Volume 1: Hieroglyphic Transliteration, Translation, and Commentary. Volume 2: Facsimile Plates and Line for Line Hieroglyphic Transliteration.

[46] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, 1996.

[47] A. Brodnik and J. M. Munro. Membership in constant time and almost-minimum space. *SIAM Journal of Computing*, 28(3):1627–1640, 1999.

[48] C. Browne. *Hex Strategy: Making the right connections*. A K Peters, 2000.

[49] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35:677–691, 1986.

[50] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):142–170, 1992.

[51] J. R. Burch, E. M.Clarke, K. L. McMillian, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[52] W. Burgard. Personal communication, 2002.

[53] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, pages 165–204, 1994.

[54] G. Cabodi, P. Camurati, and S. Quer. Improved reachability analysis of large finite state machines. In *Computer-Aided Design (CAD)*, pages 354–359, 1996.

[55] B. Chazelle. Reporting and counting segment intersections. *Computing System Science*, 32:200–212, 1986.

[56] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting lines in the plane. *Journal of the ACM*, 39:1–54, 1992.

[57] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Parallel and Distributed Information Systems (PDIS)*, 1996.

[58] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 241–257, 1996.

[59] Y. Choueka, A. S. Fraenkel, S. T. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *ACM SIGIR*, pages 88–96, 1986.

[60] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science, pages 130–142. Springer, 1997.

[61] A. Cimatti, F. Giunchiglia, and M. Roveri. Abstraction in planning via model checking. Technical report, IRST, 2000.

[62] A. Cimatti and M. Roveri. Conformant planning via model checking. In *European Conference on Planning (ECP)*, pages 21–33, 1999.

[63] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, pages 875–881, 1998.

[64] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *REX School Symposium, A Decade of Concurrency*, Lecture Notes in Computer Science, pages 124–175. Springer, 1993.

[65] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[66] E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[67] R. Cleaveland. Tableau-based model checking in the propositional $\mu$-calculus. *Acta Informatica*, 27:725–747, 1990.

[68] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite s tate verification. In *Proceedings of the 23$^{rd}$ International Conference on Software Engineering (ICSE)*, pages 37–46. IEEE Computer Society, 2001.

[69] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Converence on Software Engineering (ICSE)*, 2000.

[70] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[71] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines using symbolic execution. In *Automatic Verification Methods for Finite State Machines*, pages 365–373, 1989.

[72] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[73] A. Crauser. *External Memory Algorithms and Data Structures in Theory and Practice*. PhD thesis, MPI-Informatik, Universität des Saarlandes, 2001.

[74] J. C. Culberson. Sokoban is PSPACE-complete. In *Fun for Algorithms (FUN)*, pages 65–76. Carleton Scientific, 1998.

[75] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[76] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[77] E. D. Demaine, M. L. Demaine, and J. O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *Canadian Conference on Computational Geometry*, pages 211–219, 2000.

[78] D. C. Dennet. Minds, machines, and evolution. In C. Hookway, editor, *Cognitive Wheels: The Frame Problem of AI*, pages 129–151. Cambridge University Press, 1984.

[79] R. B. Dial. Shortest-path forest with topological ordering. *Communication of the ACM*, 12(11):632–633, 1969.

[80] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[81] D. L. Dill. *Trace Theory for Automatic Hierarchial Verification of Speed-Independent Circuits*. MIT Press, 1988. ACM Distinguished Dissertation.

[82] J. F. Dillenburg and P. C. Nelson. Perimeter search (research note). *Artificial Intelligence*, 65(1):165–178, 1994.

[83] M. B. Do and S. Kambhampati. Sapa: a domain-independent heuristic metric temporal planner. In *European Conference on Planning (ECP)*, pages 109–120, 2001.

[84] D. Dolev, M. Klawe, and M. Rodeh. An o(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 1982.

[85] P. Doucette, P. Agouris, A. Stefanidis, and M. Musavi. Self-organized clustering for road extraction in classified imagery. *Journal of Photogrammetry and Remote Sensing*, 55(5-6):347–358, March 2001.

[86] W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[87] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.

[88] J. Eckerle. *Memory-Limited Heuristic Search (German)*. PhD thesis, University of Freiburg, 1998. DISKI, Infix.

[89] J. Eckerle and T. Lais. Limits and possibilities of sequential hashing with supertrace. In *Formal Description Techniques for Distributed Systems and Communication Protocols, Protocol Specification, Testing and Verification (FORTE/PSTV)*. Kluwer, 1998.

[90] J. Eckerle and S. Schuierer. Efficient memory-limited graph search. In *German Conference on Artificial Intelligence (KI)*, pages 101–112. Springer, 1995.

[91] S. Edelkamp. Suffix tree automata in state space search. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science, pages 381–385. Springer, 1997.

[92] S. Edelkamp. *Data Structures and Learning Algorithms in State Space Search*. PhD thesis, University of Freiburg, 1999. Infix.

[93] S. Edelkamp. Directed symbolic exploration and its application to AI-planning. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 84–92, 2001.

[94] S. Edelkamp. First solutions to PDDL+ planning problems. In *Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*, pages 75–88, 2001.

[95] S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, 2001. 13-24.

[96] S. Edelkamp. Prediction of regular search tree growth by spectral analysis. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science. Springer, 2001. 154-168.

[97] S. Edelkamp. Mixed propositional and numerical planning in the model checking integrated planning system. In *International Conference on AI Planning & Scheduling (AIPS), Workshop on Temporal Planning*, 2002.

[98] S. Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *Artificial Intelligence Planning and Scheduling (AIPS)–Workshop on Model Checking*, 2002.

[99] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[100] S. Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)*, 2003. To appear.

[101] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science, pages 135–147. Springer, 1999.

[102] S. Edelkamp and M. Helmert. On the implementation of MIPS. In *Artificial Intelligence Planning and Scheduling (AIPS)–Workshop on Model Theoretic Approaches to Planning*, pages 18–25, 2000.

[103] S. Edelkamp and M. Helmert. The model checking integrated planning system MIPS. *AI-Magazine*, pages 67–71, 2001.

[104] S. Edelkamp and R. E. Korf. The branching factor of regular search spaces. In *National Conference on Artificial Intelligence (AAAI)*, 1998. 299–304.

[105] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 2002.

[106] S. Edelkamp and P. Leven. Directed automated theorem proving. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, Lecture Notes in Computer Science. Springer, 2002.

[107] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Model Checking Software (SPIN)*, Lecture Notes in Computer Science, pages 57–79. Springer, 2001.

[108] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science, pages 57–79. Springer, 2001.

[109] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

[110] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[111] S. Edelkamp and U. Meyer. Theory and practice of time-space trade-offs in memory limited search. In *German Conference on Artificial Intelligence (KI)*, Lecture Notes in Computer Science, pages 169–184. Springer, 2001.

[112] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.

[113] S. Edelkamp and F. Reffel. Deterministic state space planning with BDDs. Technical Report 121, University of Freiburg, 1999. (Summary published in *European Conference on Planning (ECP)*, Preprint, pages 381–382, 1999).

[114] S. Edelkamp and F. Reffel. Deterministic state space planning with BDDs. In *European Conference on Planning (ECP), Preprint*, pages 381–382, 1999.

[115] S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.

[116] S. Edelkamp and P. Stiegeler. Implementing HEAPSORT with $n \log n - 0.9n$ and QUICKSORT with $n \log n + 0.2n$ comparisons. *ACM Journal of Experimental Algorithmics*, 2002.

[117] D. R. Edelson and I. Pohl. A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85–102. USENIX Association, 1991.

[118] E. Emerson and J. Srinivasan. Branching time temporal logic. In *REX workshop*, Lecture Notes in Computer Science, pages 123–172. Springer, 1989.

[119] J. W. Estes. *The Medical Skills of Ancient Egypt*. Canton: Science History Publications, 1989.

[120] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1970.

[121] U. Feige. A fast randomized LOGSPACE algorithm for graph connectivity. *Theoretical Computer Science*, 169(2):147–160, 1996.

[122] U. Feige. A spectrum of time-space tradeoffs for undirected $s - t$ connectivity. *Journal of Computer and System Sciences*, 54(2):305–316, 1997.

[123] A. Felner. Finding optimal solutions to the graph-partitioning problem with heuristic search. In *Symposium on the Foundations of Artificial Intelligence*, 2001.

[124] Z. Feng and E. Hansen. Symbolic heuristic search for factored markov decision processes. In *National Conference on Artificial Intelligence (AAAI)*, 2002.

[125] J. C. Fernadez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transactions systems. *Formal Methods in System Design*, 1:251–273, 1992.

[126] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[127] P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.

[128] M. P. Fourman. Propositional planning. In *Artificial Intelligence Planning and Scheduling (AIPS)-Workshop on Model-Theoretic Approaches to Planning*, pages 10–17, 2000.

[129] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Artificial Intelligence Research*, 9:367–421, 1998.

[130] M. Fox and D. Long. The detection and exploration of symmetry in planning problems. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 956–961, 1999.

[131] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, UK, 2001.

[132] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[133] R. Fraer, G. Kamhi, M. Y. V. B. Ziv, and L. Fix. Efficient reachability analysis for verification and falsification. IBM FV'2000 Summer Seminar, 2000. www.haifa.il.ibm.com/ibmfv2000/abstracts/fraer.html.

[134] 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.

[135] A. Fukunaga, G. Tabideau, S. Chien, and C. Yan. ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *International Symposium on AI, Robotics and Automation in Space*, 1997.

[136] B. Gaines. Behaviour/structure transformations under uncertainty. *International Journal of Man-Machine Studies*, 8(3):337–365, 1976.

[137] M. Gardner. *The Mathematical Games of Sam Loyd*. Dover Publications, 1959.

[138] J. Gaschnik. *Performance Measurement and Analysis of certain Search Algorithms*. PhD thesis, Department of Computer Science, Carnigie-Mellon University, 1979.

[139] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, ETH Zürich, 1993.

[140] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1987.

[141] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *National Conference on Artificial Intelligence (AAAI)*, pages 905–912, 1998.

[142] A. Gerevini and I. Serina. Fast planning through greedy action graphs. In *National Conference of Artificial Intelligence (AAAI)*, 1999.

[143] A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs with action costs. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[144] M. Ginsberg. Step toward an expert-level bridge-playing program. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 584–589, 1999.

[145] F. Giunchiglia and P. Traverso. Planning as model checking. In *European Conference on Planning (ECP)*, pages 1–19, 1999.

[146] P. Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification (CAV)*, number 531 in Lecture Notes in Computer Science, pages 176–185. Springer, 1991.

[147] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.

[148] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[149] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[150] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer, 1987.

[151] M. G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.

[152] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[153] E. Guéré and R. Alami. One action is enough to plan. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[154] S. Gutmann. *Robust Navigation of autonomous mobile systems (German)*. PhD thesis, University of Freiburg, 1999. ISBN 3-89838-241-9.

[155] K. Haase. The intelligent database interface: Integrating AI and database systems. In *European Conference on Artificial Intelligence (ECAI)*. Wiley, 1994.

[156] S. Handley, P. Langley, and F. Rauscher. Learning to predict the duration of an automobile trip. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 219–223, New York, 1998. AAAI Press.

[157] E. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129:35–62, 2001.

[158] E. A. Hansen, R. Zhou, and Z. Feng. Symbolic heuristic search using decision diagrams. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2002.

[159] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, 1992.

[160] J. Harrison. Optimizing proof search in model elimination. In *Conference on Automated Deduction (CADE)*, pages 313–327, 1996.

[161] 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.

[162] A. C. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1990.

[163] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 140–149, 2000.

[164] P. Haslum and H. Geffner. Heuristic planning with time and resources. In *European Conference on Planning (ECP)*, pages 121–132, 2001.

[165] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.

[166] E. A. Heinz. Efficient interior-node recognition. *ICCA Journal*, 21(3):156–167, 1999.

[167] E. A. Heinz. *Scalable Search in Computer Chess*. Vierweg, 2000.

[168] M. Helmert. On the complexity of planning in transportation and manipulation domains. Master's thesis, Computer Science Department Freiburg, 2001.

[169] M. Helmert. On the complexity of planning in transportation domains. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science, pages 349–360. Springer, 2001.

[170] M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[171] D. Henrich, C. Wurll, and H. Wörn. Multi-directional search with goal switching for robot path planning. In *Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE98)*, volume 2, pages 75 – 84, 1998.

[172] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[173] I. T. Hernadvölgyi. Using pattern databases to find macro operators. In *National Converence on Artificial Intelligence (AAAI)*, page 1075, 2000.

[174] I. T. Hernadvölgyi and R. C. Holte. A space-time tradeoff for memor-based heuristics. In *National Congerence on Artificial Intelligence (AAAI)*, pages 704–709, 1999.

[175] C. A. Hipke. *Verteilte Visualisierung von Geometrischen Algorithmen*. PhD thesis, University of Freiburg, 2000.

[176] D. S. Hirschberg. A linear space algorithm for computing common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[177] J. Hoffmann. A heuristic for domain independent planning and its use in an enforced hill climbing algorithm. In *ISMIS*, Lecture Notes in Computer Science, pages 216–227. Springer, 2000.

[178] J. Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 453–458, 2001.

[179] J. Hoffmann. Extending FF to numerical state variables. In *European Conference on Artificial Intelligence*, 2002.

[180] J. Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

[181] J. Hoffmann and B. Nebel. Fast plan generation through heuristic search. *Artificial Intelligence Research*, 14:253–302, 2001.

[182] S. Hölldobler and H.-P. Stör. Solving the entailment problem in the fluent calculus using binary decision diagrams. In *Artificial Intelligence Planning and Scheduling (AIPS)-Workshop on Model-Theoretic Approaches to Planning*, pages 32–39, 2000.

[183] R. Holte and I. Hernadvolgyi. A space-time tradeoff for memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI)*, pages 704–709, 1999.

[184] M. Holzer and S. Schwoon. Assembling molecules in atomix is hard. Technical Report 0101, Institut für Informatik, Technische Universität München, 2001.

[185] G. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Conference on Protocol Specification, Testing, and Verification, INWG/IFIP*, 1992.

[186] G. J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2):32–44, 1990.

[187] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[188] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification (PSTV)*, pages 339–346. North-Holland, Amsterdam, 1993.

[189] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[190] G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Formal Description Techniques for Distributed Systems and Communication Protocols, Protocol Specification, Testing and Verification (FORTE/PSTV)*, pages 481–497, 1999.

[191] E. Horvitz and S. Zilberstein. Computational tradeoffs under bounded resources. *Artificial Intelligence*, 126:1–4, 2001.

[192] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A grandmaster chess machine. *Scientific American*, 4:44–50, 1990.

[193] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation*, pages 266–271, 1993.

[194] F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. In *German Conference on Artificial Intelligence (KI)*, pages 229–243, 2001.

[195] P. Jackson. *Introduction to Expert Systems*. Addison Wesley Longman, 1999.

[196] R. Jensen and M. M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Artificial Intelligence Research*, 13:189–226, 2000.

[197] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *National Conference on Artificial Intelligence (AAAI)*, 2002.

[198] R. Jones. *Garbage Collection*. John Wiley & Sons, 1996.

[199] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.

[200] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reachtive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[201] M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.

[202] M. Kamel and S. Leue. Vip: A visual editor and compiler for v-promela. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 471–486. Springer, 2000.

[203] P. D. Karp and S. M. Paley. Knowledge representation in the large. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 751–758. Morgan Kaufmann, 1990.

[204] H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1194–1201, 1996.

[205] H. Kautz and J. Walser. State-space planning by integer optimization. In *National Conference on Artificial Intelligence (AAAI)*, 1999.

[206] J. Kelly. *Artificial Intelligence: A Modern Myth*. Ellis Horwood, 1993.

[207] J. Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, pages 1340–1330, 1987.

[208] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Special Issue of Journal of Computer and System Sciences on selected papers of STOC 1994*, 55(1):3–23, 1997.

[209] C. Knoblock. Generating parallel execution plans with a partial order planner. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 98–103, 1994.

[210] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching*. Addison-Wesley, 1973.

[211] J. Koehler. Elevator control as planning problem. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 331–338, 2000.

[212] J. Koehler, B. Nebel, and Y. Dimopoulos. Extending planning graphs to an adl subset. In *European Conference on Planning (ECP)*, pages 273–285, 1997.

[213] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[214] R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[215] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.

[216] R. E. Korf. From approximate to optimal solutions: A case study of number partitioning. In *National Converence on Artificial Intelligence (AAAI)*, pages 266–272, 1998.

[217] R. E. Korf. Divide-and-conquer bidirectional search: First results. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1184–1191, 1999.

[218] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 2001.

[219] R. E. Korf and M. Reid. Complexity analysis of admissible heuristic search. In *National Conference on Artificial Intelligence (AAAI)*, 1998. 305–310.

[220] R. E. Korf, M. Reid, and S. Edelkamp. Time Complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1–2):199–218, 2001.

[221] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *National Conference on Artificial Intelligence (AAAI)*, pages 1202–1207, 1996.

[222] R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.

[223] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD Explorations*, 2(1):1–15, 2000.

[224] D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[225] O. Krieger. *HFS: A flexible file system for shared-memory multiprocessors*. PhD thesis, University of Toronto, 1994.

[226] H. W. Kuhn. The Hungarian method for the assignment problem. In *Naval Res. Logist. Quart*, pages 83–98, 1955.

[227] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Sympostium on Parallel and Distributed Processing*, pages 169–177. IEEE, 1996.

[228] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, 1998.

[229] J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and ressources. In *European Conference on Artificial Intelligence (ECAI)*, pages 501–505, 2000.

[230] U. D. Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *National Conference on Artificial Intelligence (AAAI)*, 2002.

[231] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. In *Advances in Computer Chess VII*, pages 135–162. University of Limburg, Maastricht, Netherlands, 1994.

[232] P. Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.

[233] K. G. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer, 1995.

[234] P.-A. Larson. Dynamic hashing. *BIT*, 18(2):184–201, 1978.

[235] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, Apr. 1992.

[236] A. Levy and D. S. Weld. Intelligent internet systems. *Artificial Intelligence*, 118:1–14, 2000.

[237] O. Lichtenstein and A. Pneuli. Checking that finite state concurrent programs satisfy their linear specification. In *ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, 1985.

[238] F. Lin. System r. *AI-Magazine*, pages 73–76, 2001.

[239] F. J. Lin, P. M. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *ACM*, pages 126–135, 1988.

[240] W. Litwin. Virtual hashing: a dynamically changing hashing. In *Very Large Databases*, pages 517–523, 1978.

[241] A. Lluch-Lafuente, S. Leue, and S. Edelkamp. Partial order reduction in directed model checking. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science, pages 112–127. Springer, 2002.

[242] M. Löbbing and I. Wegener. The number of night's tours equals 33,439,123,484,294 - counting with binary decision diagrams. *The Electronic Journal of Combinatorics*, 1996.

[243] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Artificial Intelligence Research*, 10:87–115, 1998.

[244] D. Long and M. Fox. Automatic synthesis and use of generic types in planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 196–205, 2000.

[245] D. Long and M. Fox. Encoding temporal planning domains and validating temporal plans. In *Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*, 2001.

[246] D. Long and M. Fox. Hybrid stan: Identifying and managing combinatorial optimisation sub-problems in planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 445–452, 2001.

[247] U. Lorenz. Controlled conspriracy-2 search. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 466–478, 2000.

[248] D. W. Loveland. Theorem-provers combining model elimination and resolution. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 73–86. University Press, Edinburgh, 1969.

[249] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Symposium on Math, Statistics, and Probability*, volume 1, pages 281–297, 1967.

[250] N. R. Mahapatra and S. Dutt. Scalable global and local hashing strategies for duplicate pruning in parallel A* graph search. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):738–756, 1997.

[251] S. M. Majercik and M. L. Littmann. Using caching to solve larger probabilistic planning problems. In *National Conference on Artificial Intelligence (AAAI)*, pages 954–959, 2000.

[252] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[253] T. Marsland and J. Schaeffer, editors. *Chess, Computers, and Cognition*. Springer, 1990.

[254] T. A. Marsland. *Ecyclopedia of Artificial Intelligence*, chapter Computer Chess and Search, pages 224–241. Wiley & Sons, 1992.

[255] D. A. McAllester. Conspiricy-number-search for min-max searching. *Artificial Intelligence*, 35:287–310, 1988.

[256] D. A. McAllester. Automatic recognition of tractability in inference relation. *Journal of the ACM*, 40(2):284–303, 1993.

[257] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[258] D. McDermott. The 1998 AI Planning Competition. *AI Magazine*, 21(2), 2000.

[259] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[260] D. McVitie and L. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–492, 1971.

[261] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, 2002.

[262] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[263] C. Meinel and C. Stangier. Hierarchical image computation with dynamic conjunction scheduling. In *International Conference on Computer Design (ICCD)*, 2001.

[264] M. L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report A.I. MEMO 58 Rvsd. and MAC-M-129, Cambridge, Massachusetts, 1963.

[265] T. Minura and T. Ishida. Stochastic node caching for efficient memory-bounded search. In *National Conference on Artificial Intelligence (AAAI)*, pages 450–459, 1998.

[266] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[267] D. E. Muller, A. Saoudi, and P. E. Schnupp. Alternating automata. The weak monadic theory of the tree, and its complexity. In L. Kott, editor, *International Colloquium on Automata, Languages and Programming*, pages 275–283. Springer, 1986.

[268] M. Müller. *Computer Go as a sum of local games*. PhD thesis, ETH Zürich, 1995.

[269] M. Müller. Partial order bounding: A new approach to game tree search. *Artificial Intelligence*, 2001.

[270] M. Müller. Proof set search. Technical Report TR01-09, University of Alberta, 2001.

[271] M. Müller and T. Tegos. Experiments in computer amazons. *More Games of No Chance, Cambridge University Press*, 2001.

[272] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 87–88, 2001.

[273] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender:. Complexity of finite-horizon markov decision process problems. *Journal of the ACM*, 4:681–720, 2000.

[274] Y. Murase, H. Matsubara, and Y. Hiraga. Automatic making of Sokoban problems. In *Pacific Rim Conference on AI*, 1996.

[275] C. Navigation Technologies, Sunnyvale. Software developer's toolkit, 5.7.4 solaris edition, 1996.

[276] B. Nebel. Personal communication, 2002.

[277] A. Newell, V. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proceedings ICIP*, 1959.

[278] N. J. Nilsson. *Principles of Artificial Intelligence*. Symbolic Computation. Springer, 1982.

[279] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[280] C. Okasaki. *Purely Functional Data Structures*, chapter 3. Cambridge University Press, 1998.

[281] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking and model checking. In *Lectoure Notes in Computer Scence (1102)*, pages 411–414, 1996.

[282] R. Parekh, C. Nichitiu, and V. Honovar. A polynomial time incremental algorithm for regular grammar inference. Technical Report 97-03, Department of computer science, Iowa State University, 1997.

[283] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, GI Conference*, Lecture Notes in Computer Science, pages 167–183. Springer, 1981.

[284] B. W. Parkinson, J. J. Spilker, P. Axelrad, and P. Enge. *Global Positioning System: Theory and Applications*. American Institute of Aeronautics and Astronautics, 1996.

[285] E. Pattison-Gordon. Thenetsys: A semantic network system. Technical Report DSG-93-02, Decision Systems Group, Brigham and Womens Hospital, Boston, 1993.

[286] L. C. Paulson. Strategic principles in the design of Isabelle. In *Proceedings of the CADE Workshop on Strategies in Automated Deduction*, pages 11–16, 1998.

[287] J. Pearl. *Heuristics*. Addison-Wesley, 1985.

[288] E. P. D. Pednault. Formulating multiagend, dynamic-world problems in the classical framework. In *Reasoning about Action and Plans*, pages 47–82. Morgan Kaufmann, 1986.

[289] E. Pednould. ADL: Exploring the middleground between Strips and situation calculus. In *Knowledge Representation (KR)*, pages 324–332. Morgan Kaufman, 1989.

[290] D. A. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in Systems Design*, 8:39–64, 1996.

[291] D. A. Peled. Ten years of partial order reduction. In *Computer-Aided Verification (CAV)*, number 1427 in Lecture Notes in Computer Science, pages 17–28. Springer, 1998.

[292] P. Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, 1999.

[293] L. Piegl and W. Tiller. *The nurbs book*. Springer, 1997.

[294] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[295] I. Pohl. Practical and theoretical considerations in heuristic search algorithms. *Machine Intelligence*, 8:55–72, 1977.

[296] E. L. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215, 1943.

[297] C. A. Pribe and S. O. Rogers. Learning to associate driver behavior with traffic controls. In *Proceedings of the 78th Annual Meeting of the Transportation Review Board*, Washington, DC, January 1999.

[298] M. L. Puterman. *Markov Decision Processes*. Wiley Series in Probability, 1994.

[299] B. Rahardjo. Spin as a hardware design tool. In *First SPIN Workshop*, 1995.

[300] A. Rapoport. *Two-Person Game Theory*. Dover, 1966.

[301] D. Ratner and M. K. Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.

[302] I. Refanidis and I. Vlahavas. Heuristic planning with ressources. In *European Conference on Planning (ECAI)*, pages 521–525, 2000.

[303] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods (FM)*, pages 195–211, 1999.

[304] P. Regnier and B. Fade. Détermination du parallélisme maximal et optimisation temporelle dans les plans d'actions linéaires. *Revue d'intelligence artificielle*, 5(2):67–88, 1991.

[305] A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[306] A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[307] J. Rintanen and H. Jungholt. Numeric state variables in constraint-based planning. In *European Conference on Planning (ECP)*, Lecture Notes in Computer Science, pages 109–121. Springer, 1999.

[308] 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.

[309] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Computer-Aided Design (CAV)*, pages 139–144. IEEE, 1994.

[310] T. A. Runkler, editor. *Information Mining (German)*. Vierweg, 2000.

[311] S. Russell. Efficient memory-bounded search methods. In *European Conference on Artificial Intelligence (ECAI)*, pages 1–5. Wiley, 1992.

[312] A. Samuel. Some studies in machine learning using the game of checkers- recent progress. *IBM Journal of Research and Development*, pages 210–229, 1959.

[313] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[314] J. Schaeffer. Conspiracy numbers. *Advances in Computer Chess 5*, pages 199–217, 1989.

[315] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer, 1997.

[316] J. Schaeffer. The games computer (and people) play. *Advances in Computers 50*, pages 189–266, 2000.

[317] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–290, 1991.

[318] J. C. Schlimmer and L. A. Hermens. Software agents: Completing patterns and constructing user interfaces. *Journal of Artificial Intelligence Research*, 1:61–89, 1993.

[319] P. D. A. Schofield. Complete solution of the eight puzzle. In *Machine Intelligence 2*, pages 125–133. Elsevier, 1967.

[320] S. Schroedl. *Negation as Failure in Explanation-Based Generalization*. PhD thesis, Computer Science Department Freiburg, 1998. Infix, 181.

[321] S. Schroedl, S. Rogers, and C. Wilson. Map renfinement from GPS traces. Technical Report RTC 6/2000, DaimlerChreisler Research adn Technology North America, Palo Alto, CA, 2000.

[322] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, 5(12), 2000.

[323] F. Schulz, D. Wagner, and C. Zaroliagis. Using multi-level graphs for timetable information. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2002.

[324] A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 297–302, 1989.

[325] E. Y. Shapiro. *Algorithmic Program Debuggging*. PhD thesis, Yale University, 1983. Published under the same title by MIT press.

[326] H. A. Simon and A. Newell. Heuristic problem solving: The next advance in operation research. *Operations Research*, 6, 1958.

[327] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, pages 119–153, 2001.

[328] D. J. Slate. Interior-node score bounds in a brute-force chess programm. *ICCA Journal*, 7(4):184–192, 1984.

[329] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[330] F. Somenzi and R. Bloem. Efficient buchi automata from LTL formulae. In *Computer Aided Verification (CAV)*, 2000.

[331] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker Verlag, Aachen, 1996.

[332] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. In *Conference on Automated Deduction (CADE)*, volume 230, pages 573–587. Springer, 1986.

[333] C. Stirling and D. Walker. Local model checking in the modal $\mu$-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[334] K. Stoffel, M. Taylor, and J. Hendler. Integrating knowledge and data-base technologies. Technical report, University of Maryland, 1996.

[335] M. M. Syslo, N. Deo, and J. S. Kowalik. *Discrete Optimization Algorithms with Pascal Programs*. Prentice-Hall, 1983.

[336] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[337] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 756–761, 1993.

[338] G. Tesauro. Temporal difference learning and TD-Gammon. *Communication of the ACM*, 38(3):58–68, 1995.

[339] L. Thoma and N. Zeh. I/O-efficient algorithms for sparse graphs. In *Memory Hierarchies*, Lecture Notes in Computer Science. Springer, 2002. To appear.

[340] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[341] A. Turing. Computing machinery and intelligence. *Mind*, 59, 1950.

[342] A. Valmari. A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–165, 1991.

[343] H. J. van den Herik. Strategy in chess endgames. *Computer Game-Playing: Theory and Practice*, pages 87–105, 1983.

[344] M. M. Veloso, M. A. Pé, and J. G. Carbonell. Nonlinear planning with parallel resource allocation. In *Innovative Approaches to Planning, Scheduling and Control*, pages 207–212, 1990.

[345] W. Visser and H. Barringer. CTL* model checking for SPIN. *Software Tools for Technology Transfer*, 2(4), 2000.

[346] N. Voelker, 2002. Personal Communication.

[347] J. Wang, S. Rogers, C. Wilson, and S. Schroedl. Evaluation of a blended DGPS/DR system for precision map refinement. In *Proceeedings of the ION Technical Meeting 2001*, Institute of Navigation, Long Beach, CA, 2001.

[348] G. I. Webb. Efficient search for association rules. In *Knowledge Discovery and Data Mining*, pages 99–107, 2000.

[349] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 88–97. Springer, 1998.

[350] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 59–70. Springer, 1993.

[351] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. Hallaron. Space- and time-efficient BDD construction via working set control. In *Asia and South Pacific Design Automation*, pages 423–432, 1998.

[352] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.

[353] W. Zhang. Depth-first branch-and-bound versus local search. In *National Conference on Artificial Intelligence (AAAI)*, pages 930–935, 2000.

# Index


$(n^2 - 1)$ Puzzle, 6, 56
$\alpha\beta$ pruning, 14
$k$-means algorithm, 333
$t$-limited scheme, 80

A*, 6, 9, 31, 36, 56, 90
A*+INDFS, 250
Absence, 225
abstract planning problem, 144
accepting run, 251
action graph, 190
action inference, 177
action planning, 126, 196
action-based exploration, 96
active, 313
activeness bonus, 313, 314
ADD, 13
additive pattern database, 145
admissibility, 145
admissible, 6, 73, 197, 338
algebraic multiplicity, 58
algorithm agglom, 333
algorithm Apriori, 17
alive state, 214
American Checkers, 6
ample set, 277
anomaly in depth-bounded search, 265
approximation error, 331
arrangement, 322
Artificial Intelligence, 17
asymptotic branching factor, 36
atom, 131
Atomix, 65, 72
auto tactic, 201
automated theorem proving, 17, 196
autonomous spacecraft control, 11

B-spline, 330
Büchi automaton, 251
Backgammon, 6
backward edge, 279
backward move, 80
bad sequence, 269
basis, 57
basis-transformation, 57
BDD, 12, 123, 291
belief space, 11
Bellman approximation, 157
best-first, 196
best-first search, 6
BFS, 6
bit-state hashing, 8, 79, 240
Blackbox, 124
blast tactic, 201
blind search strategy, 263
block, 278
blocked, 239
Blocks World, 11, 99, 100
bootstrapping, 10
boundary cluster, 328
bounded model checking, 14, 293
branching factor, 14
Bridge, 6
brute-force branching factor, 48
brute-force search tree, 47
bubble-sort, 213
buffer, 24
buffer tree, 9

capacity constraint, 138
cardinality constraint, 99
case-based reasoning, 17
characteristic equation, 58
characteristic function, 114
checkpoint, 24
Chess, x, 6
Chinook, 16
chord length, 331
cluster seed location, 326
combinatorial game theory, 15

commutativity of operators, 177
complete validation, 224
component tree, 337
computational complexity, 322
Computer Amazons, 6
conformant planning, 11, 13, 113, 155, 191
conjunctive partitioning, 12
consistent, 6, 42, 197, 294, 310, 338
conspiracy number search, 15
constant predicate, 93, 114
CORBA, 31
coverage problem, 5
curvature, 332
curvature error, 332

dangerous state, 239
data mining, 16
dead-end, 119
dead-end recognition table, 10
deadlock, 239
dependency relation, 174
dependent operators, 177
depth-first search, 265
Depth-First-Branch-and-Bound, 312
Desert-Rats, 175, 180
DFBnB, 90
diagonizability, 61
diagonizable, 58, 61
diagonizable matrix, 57
direct variable conflict, 174
directed automated theorem proving, vii, 195
directed model checking, viii, 14, 244, 261, 262, 270
directed stochastic model checking, 259
direction, 76
Discoplan, 99, 100
disjoint pattern databases, 134, 145
disjunctive partitioning, 12
domain abstraction, 133
duplicate state, 295
duplicates, 56

effective search tree depth, 150
eigenvalue, 58
eigenvector, 58
elevator scheduling, 11
elimination rule, 203
enabled set, 277
enabled transition, 277
enabledness of operator, 177
end user programming, 212
endgame database, 15
endomorphism, 57
enforced hill climbing, 112, 120, 207
equilibrium distribution, 44
equilibrium fraction, 38
equivalence graph, 59
error detection, 14
event queue, 323
execution cost, 190
expert system, 4
exploratory mode, 257, 275

fact, 131
fact-space exploration, 96, 120
fault-finding mode, 257, 275
fertile node, 45
FF-heuristic, 112, 125
fitness, 256
forward pruning, 14
forward set simplification, 116
Four Connect, x, 15
frame, 147
frame problem, 4
frontier search, 9
FSM pruning, 8, 90, 139
fuel constraint, 138
full-accepting, 252, 253
full-state trail-directed search, 268
fully accepting, 249
fully expanded, 277
fully-accepting, 253
function expand, 6
function merging mechanism, 212

game-theoretical value, 15
garbage collector problem, 4
general position, 322
General-Node-Ordering A*, 312, 314
generalized move, 77
generic type, 14, 112
geometric multiplicity, 58
GIOP, 31
Go, 15

goal nodes, 309
goal ordering cut, 130
goal state, 75
GoMoku, 15
grammar inference, 214
graph algorithm, 322
graph extension, 11, 130
graph partitioning, 6
Graphplan, 11, 12, 100, 130, 174, 177, 178, 189
Grid, 99
Gripper, 99, 123, 138
ground entailment problem, 208
grounded planning instance, 166
grounded predicate, 131
grounded propositional planning problem, 143

Hamming distance, 266
hash compaction, 80
hash table, 82
heading information, 329
Heap-Of-Heaps, 309, 313
helpful action cut, 130
helpful actions, 120
heuristic, 309
heuristic branching factor, 47
heuristic pattern database, 118
heuristic search, x, 6, 113, 126
heuristic symbolic search, 155
Hex, x, 6, 15
Hexy, 15
hierarchical memory, 4
HSP-heuristic, 112

IDA*, 6, 9, 20, 36, 56, 90, 308
identity invariant, 99, 100
IID, 215
independent abstraction set, 134
independent operators, 177
INDFS, 249
indirect variable conflict, 175
inductive logic programming, 212
information mining, 16
information theoretical bound, 8
informed search, 121
inherently sequential domain, 175
intelligent internet system, 16
interior-node recognition, 16
invisible transition, 278
Isabelle, vii, 195, 196
ISO Reference Model, 27
iterative deepening, 14, 196

Jordan normal form, 57, 66
Jugs-and-Water, 175

Kalman filter, 325

language accepted by Büchi automaton, 251
line segment, 321
linear mapping, 57
live-complete set of strings, 214
local model checking, 292
Localized A*, 339
Logistics, 11, 99, 123, 124
lower bound, 294
LTL, 226

macro recorder, 212
main memory, ix, 307
main vector, 66
maintenance mode, 257
make-span, 189
map, 320
material signature, 16
max-atom heuristic, 155
max-pair heuristic, 112, 126, 131
MDP, 11, 192
MEIDA*, 9
membership invariant, 99, 100
memory locality, 308
memory sensitive search algorithms, 90
memory-limited search, 9
memory-restricted search algorithm, 20, 308
merit, 309
metric planning, 11
Metric-FF, 163
mini-max game tree, 14
mini-max search, 14
minimal-window search, 14
minimum cost perfect matching, 82
minimum space algorithm, 21
MIPS, 124, 136

mobile analysis, 99
model checking, 12, 113, 126, 160, 196
Model Inference System, 212
monotone heuristic, 77
move ordering, 14, 311
move sequence, 72
Movie, 99
Mprime, 99
MREC, 9, 308
multi-player game, 14
multipath error, 325
multiple sequence alignment, 6
mutex group, 131
mutex relation, 174
Mystery, 99, 130, 138

negotiable game, 14
never claim, 225, 226, 251
neverstate, 249
Nim, x
Nine-Men-Morris, 15
non-accepting, 249, 252, 253
non-deterministic planning, 113, 155, 191
normal form, 57
NP, 214
number partitioning, 6
numerical constraint satisfaction, 167
numerical effect, 167
numerical planning, 11
numerical precondition, 167
numerical vector update, 167

object transposition, 183
on-the-fly instantiation, 169
one-way predicate, 93
operator duration, 175
operator normal form, 167
optimal solution, 333
output-sensitive algorithm, 322
overall distribution, 43, 44

page, 5, 308
page fault, 308
page frame, 308
paging strategy, 308
parallel execution time, 175
parallel plan, 175
parametric form, 330
partial observable planning, 11
Partial IDA*, 30, 79
partial observable planning, 155, 191
partial order bounding, 15
partial order reduction, 14, 293
partial search, 8
partial-accepting, 252–254
partial-state trail-directed search, 268
partially accepting, 249
partition, 146
pattern search, 139
pattern database, 12, 43
pattern databases, 90
PDB, 134
PDDL, 11
perimeter search, 10
persistent search tree, 13
PERT, 13
PERT scheduling, 176
piecewise linear, 332
plan abstraction, 156
plan relaxation, 156
planning, 113
planning pattern database, 133
planning space abstraction, 133
planning space partition, 146
position, 75
power iteration, 63
precedence graph, 190
probabilistic planning, 11, 156, 192
proof tactic, 196
proof number search, 15
proof state, 17
proof-finding, 208
proof-refinement, 208
propositional planning, 11
protocol validation, 27
protocol verification, 79
pruning set, 185
pure heuristic search, 113
push-button, 196

qualification problem, 4
quiescence search, 14

radix heap, 337
re-opening, 6, 264

reachability analysis, 130
reachable fact, 95
refinement-depth, 297
regular grammar, 214
relational product, 91, 114
relaxed planning heuristic, 11
relevance cut, 139
Response, 225, 250
response pattern, 248
restriction, 143
retrograde analysis, 16
revisiting, 266
rigid ancestor pruning, 201
road centerline, 325, 326, 329
road segment, 324
robot navigation, 11
robot-arm motion planning, 6
route planning, 6
Rubik's Cube, 12, 37, 56, 65
rule induction, 16
rule subset, 196
run of a Büchi automaton, 251

SAT, 293
Satplan, 11, 12, 14, 100, 130, 189
scanning complexity, 5, 339
schedule, 175
search algorithm, 265
search cost, 190
search horizon, 12
secondary memory, 4
segment, 327
sequential hashing, 28
sequential plan, 143, 174
Settlers, 163, 188, 190
shape point, 324
similar matrices, 57
single-atom heuristic, 155
SMA*, 9, 308
snake, 328
Software Verfication, xi
Sokoban, 10, 65, 130, 138
solution, 76
solution extraction, 11, 130
solution preserving, 144
sorting complexity, 5, 339
spatial clustering procedure, 326
spatial partitioning, 8
spectral radius, 64
SPIN, 31
SSSP, 6
stack, 145
Stan, 99
start node, 309
start state, 75
state, 75, 132
state invariant, 99
state compaction, 8
state compression, 7
state explosion problem, 160, 274
state space abstraction, 133
state space search, 27, 72
state table, 82
state-action table, 11
StaticBdd, 162
status structure, 323
sterile node, 45
sticky, 281
Strips planning problem, 11
strong cyclic plan, 126
strong planning, 11
stuttering equivalence, 278
suffix list, 7, 23
supertrace algorithm, 28
sweep-line, 322
symbolic best-first search, 122
symbolic exploration, 12
symbolic hill-climbing, 126
symbolic model checking, 14
symbolic representation, 146
symbolic weighted A*, 126
symmetry reduction, 185
synchronous product, 252

target enlargement, 125
target enlargements, 293
temporal planning, 11
the brute-force search tree, 47
theorem proving, 196
Tic-Tac-Toe, x
Tim, 99, 100
timed line segment, 321
timed path, 321
timed point, 321

Tinker, 213
tournament tree, 9
trace, 321
trace generalization, 212
trace generation, 212
trace graph, 321
tracks, 293
traditional STRIPS planning, 113
trail-directed model checking, viii, 258, 261, 270, 271
trail-directed search, 31
transient, 248
transition frequency, 334
transition relation, 114, 147
transition set, 214
transposition table, 7, 30, 79, 90
travel graph, 322
travel graph inference problem, 320
tree expansion, 36
tridiagonizability, 57
trie, 216
two-payer zero-sum game, 14

uniform, 331
unique state invariant, 99, 100
universal hashing, 29
universal planning, 113

valid endstate, 237
valid STRIPS plan, 177
Validation with guided search, 293
VepaDomain, 188
VepaServer, 187
virtual memory management, ix, 307

web mining, 16
weighted least squares fit, 330
weighted transition relation, 117
working memory, 308

Zeno-Travel, 164, 169, 171, 172, 175, 181, 182, 184, 186