TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Bildverstehen und wissensbasierte Systeme
Institut für Informatik

Representation and parallelization techniques for classical planning

Tim-Christian Schmidt

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Felix Brandt
Prüfer der Dissertation:	1. Univ.-Prof. Michael Beetz, Ph.D.
	2. Univ.-Prof. Dr. Bernhard Nebel, Albert-Ludwigs-Universtiät Freiburg

Die Dissertation wurde am 20.09.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.02.2012 angenommen.

Abstract

Artificial Intelligence as a field concerns itself with the study and design of intelligent agents. Such agents or systems perceive their environment, reason over their accumulated perceptions and through this reasoning, derive a course of action which achieves their goals or maximizes their performance measure. In many cases this planning process boils down to a principled evaluation of potential sequences of actions by exploring the resulting anticipated situations of the agent and its environment, that is to some form of combinatorial search in an implicit graph representing the interdependencies of agent actions and potential situations.

This thesis focuses on the search algorithms that implement this reasoning process for intelligent agents. The first part covers a novel subclass of such reasoning problems where a goal driven agent must find a sequence of actions to achieve its goal such that some value function of the sequence is closest to a reference value. Such problems occur for example in recommender systems and self-diagnosing agents. The contribution here is an algorithm and a family of heuristic functions based on memoization that improves on the current state-of-the-art by several orders of magnitude.

The second part focuses on cost- and step-optimal planning for goal-driven agents in which the problem is to derive the least-cost action sequence which achieves the agent's goals. Here, dynamic programming can be employed to avoid redundant evaluations. The contributions of this thesis address two challenges associated with the use of dynamic programming algorithms.

The first challenge is the memory consumption of these techniques. At the heart of such search algorithms lies a memoization component that keeps track of all generated situations and the best known way to attain them which is used to derive potential new situations. In practice the rapid growth of this component is the limiting factor for the applicability of this class of algorithms, relegating their use to problems of lower complexity or instances where the necessary graph traversal can be limited by other means such as very strong but often computationally expensive heuristics. The contribution in this thesis is a data structure that significantly reduces the necessary amount of memory to represent such a memoization component and which can be integrated into a wide range of search algorithms.

Another challenge lies in the parallelization of such algorithms to take advantage of concurrent hardware. Conceptually their soundness relies on maintaining a coherent memoization state across all participating threads or processes. As a state-of-the-art planner can explore on the order of millions of planning states per second, each of which have to be tested against and potentially update said state, standard synchronization primitives generally result in prohibitive overhead. The second contribution describes a technique that allows to adaptively compartmentalizing this state based on problem structure guaranteeing consistency across participating threads with negligible synchronization overhead. Both techniques are applicable to a wide range of search algorithms. In combination they allow to exploit the significant computational advantages of dynamic programming on problems of higher complexity while profiting from the inherent parallelism in current and future computing platforms.

Kurzfassung

Intelligente Software-Agenten zeichnen sich durch die Fähigkeit aus durch Schlussfolgerungsprozesse und Umgebungswissen eigenständig Aktionen auszuwählen und durchzuführen, die geeignet sind ihre vorgegebenen Ziele zu erreichen. In vielen Fällen sind diese Schlussprozesse im Kern als heuristische Suche über einem Umgebungs- und Aktionsmodell implementiert, wobei die Skalierbarkeit dieser Suche sowohl die mögliche Umgebungskomplexität als auch die Qualität des Agentenverhaltens direkt beeinflusst. Diese Dissertation greift diese Herausforderung auf und beschreibt im ersten Teil ein neuartiges Suchverfahren für eine Klasse von komplexen Suchproblemen die im Kontext selbstdiagnostizierender Agenten auftreten und im zweiten Teil eine neuartige Datenstruktur und Parallelisierungstechnik für auf Dynamischer Programmierung basierender Suchverfahren, die ihren Einsatz in komplexeren Umgebungen ermöglicht.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Since its inception as a scientific field, Artificial Intelligence (AI) has concerned itself primarily with the study and design of intelligent agents [McC59]. In AI, an agent is commonly defined as some entity that „perceives and acts in an environment" [RNC+10]. Embodiments of agents hence span a large range from sentient, biological beings over robots acting in a physical environment to immaterial software components interacting with their virtual peers. But what then constitutes an intelligent agent or intelligence per se? There is little agreement on a general, precise definition. In their public statement "Mainstream Science on Intelligence"[1], a group of researchers in fields related to intelligence testing gave this rather general definition:

> "[Intelligence is...] A very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience. It is not merely book learning, a narrow academic skill, or test-taking smarts. Rather, it reflects a broader and deeper capability for comprehending our surrounding - "catching on," "making sense" of things, or "figuring out" what to do".

Even when limiting oneself to definitions given by practitioners of their own field, there is considerable divergence. However, four general approaches to defining artificial intelligence are prevalent (c.f. [RNC+10]).

THINKING HUMANELY based on the idea of cognitive modeling, i.e. developing a theory of the mind and expressing it as a computer program.

> "The exciting new effort to make computers think ... machines with minds, in the full and literal sense" [Hau89]

[1] Wall Street Journal on December 13, 1994, see also [Got97]

> "[The automation of] activities that we associate with human thinking, such as decision making, problem solving, learning ..." [Bel78]

THINKING RATIONALLY based on the idea of using formal principles of valid inference and reasoning (i.e. based on "laws of thought")

> "The study of mental faculties through the use of computational models" [Cha85]

> "The study of computations that make it possible to perceive, reason and act." [Win92]

ACTING HUMANELY based on the idea of equivalence between agent and human behavior for an external observer (c.f. the "Turing Test" [Tur50])

> "The art of creating machines that perform functions that require intelligence when performed by people." [KPSM90]

> "The study of how to make computers do things at which, at the moment, people are better." [RK91]

ACTING RATIONALLY based on the idea of agents acting in a way that results in the best (expected) outcome.

> "Computational Intelligence is the study of the design of intelligent agents." [PMG98]

> "AI ... is concerned with intelligent behavior in artifacts." [Nil98]

While all these definitions have their rationale and merits, for the purposes of this thesis, I restrict this discussion to the notion that an intelligent agent takes rational actions as it relates straightforwardly to the potential merits of deploying such agents in practice. The behavior of agents, in other words the mapping from sequences of perceptions to actions is referred to as the agent function. Under this definition, it is this agent function that epitomizes its intelligence. Russel and Norvig define such rational agents as follows:

> "For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has." [RNC+10]

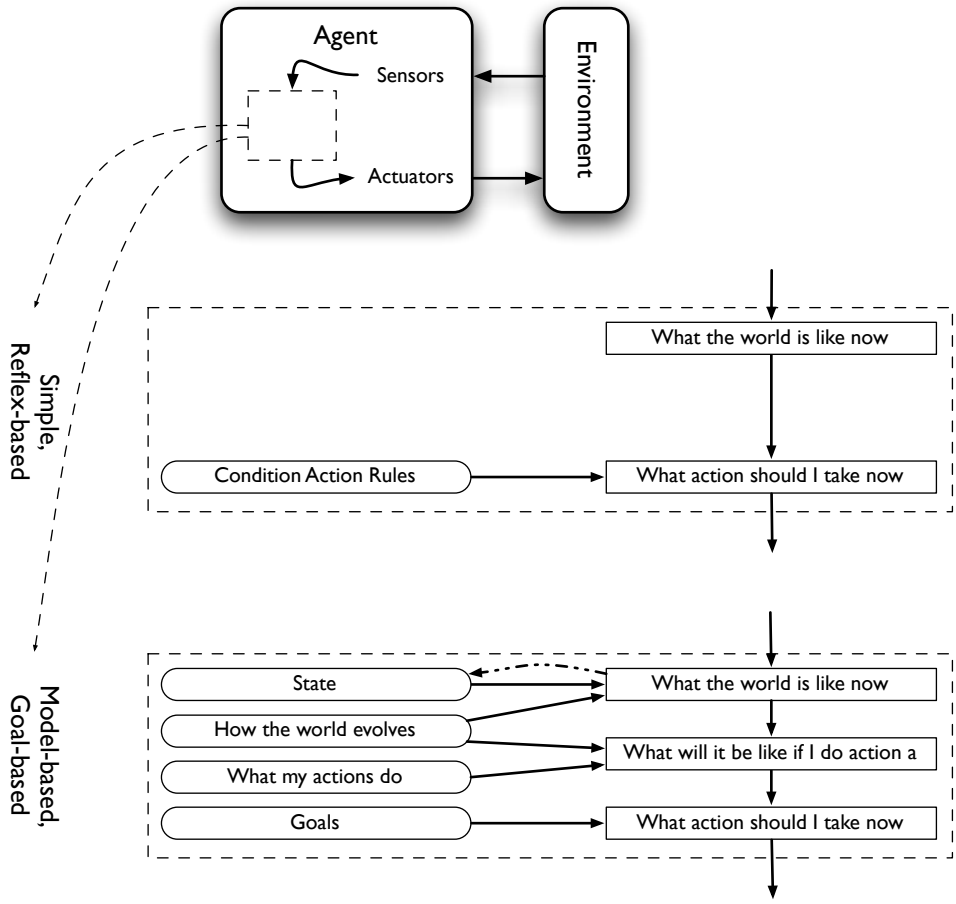


FIGURE 1.1 Schematic diagrams of a simple, reflex-based and model-based, goal-based intelligent agents according to the classification of [RNC+10]. Agent knowledge is represented as ellipses, activities as boxes.

In other words, rational action at any point in time depends on four things: (1) A performance measure defining some success criterion, (2) the agent's prior knowledge about the environment, (3) the agent's abilities to interact with the environment and (4) the hitherto accumulated perceptions about the environment. The component that implements the agent function is commonly referred to as the *agent program*.

They continue by outlining two main criteria for classifying agent programs: The first relates to the world view of the agent. For *simple* agent programs, the agent's world view corresponds to the last received perception. For *model-based* agents, the world view of the agent depends on all hitherto received perceptions and/or prior knowledge about the environment and its dynamics (i.e. the agent builds and updates an internal model of the environment). The second pertains to the action selection conditioned on that world view. *Reflex* agent programs com-

prise of a static set of condition-action rules. *Goal-based* agent programs *reason* about their world view (and often their prior knowledge of the world) to select an action to best achieve their goal. Finally *utility-based* agent programs allow for the most general behavioral patterns required for autonomous agents. Their behavior results from weighing possible (courses of) actions and their (anticipated) consequences by some unified utility measure and acting in way that (expectedly) maximizes accumulated utility over their lifetime. Figure 1.1 gives schematic examples of two prominent agent classes.

1.1 On AI planning

The facet of intelligence I want to concern myself with throughout this thesis is planning, the "figuring out, what to do" part of the agent program. Due to its importance for the design of intelligent agents, it comprises one of the central sub-disciplines of AI. Planning can be described as the process of notional anticipation of the consequences of (sequences of) actions (by an agent) in order to achieve some goal. A planning problem then is to find (or decide on) a course of action that achieves a given goal when implemented in some specific situation. Depending on the problem, these courses of actions or *plans* can take the shape of action sequences or reactive policies where later actions depend on the outcome of earlier ones. In general plans are structured (or complex) solutions and that structure (e.g. the length of the plan) has to be discovered and optimized as part of the process, which distinguishes planning from other classification and control problems. As this rather vague description already suggests, in practice planning problems and processes vary significantly depending on the intelligent agent's environment, abilities and tasks. For example environments can be fully or partially observable (e.g. the agent's sensors (and prior knowledge) allow for an accurate representation of all task relevant phenomena of the environment versus only allowing for, say, a probability over multiple possible environment states), static or dynamic (e.g. all changes to the environment are due to agent actions versus the presence of other interacting processes in the environment) and the agent's actions' effects in the environment can be deterministic or not (e.g. actuators can fail to execute a desired action properly). Over the years, a correspondingly wide variety of formal frameworks have been developed to enable formalizing of and reason in such domains.

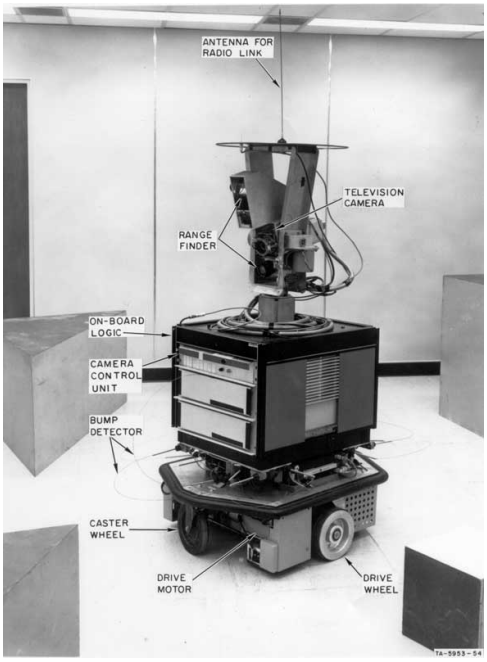


FIGURE 1.2 Stanford Research Institute's Shakey autonomous robot, the first intelligent agent that could reason about its abilities and its environment to infer suitable courses of actions in order to solve complex problems, hence melding logical reasoning and physical actions. The non-flattering name allegedly stemmed from the jerkiness of his movements. It is the oldest real inductee in the robot hall of fame and today on display at the Computer History Museum in Mountain View, California.

FIGURE 1.3 Two autonomous vehicles at the 2007 DARPA Urban Challenge navigating a four-way intersection.

1.2 Genealogy

The first agent able to comprehend its environment and abstractly reason about its abilities to plan a course of action in order to solve complex problems was Shakey (see Figure 1.2), a robot developed at the Stanford Research Institute between 1966 and 1972 [NRR+68]. The system could (on "a good day") formulate and execute plans encompassing moving from place to place and pushing blocks to satisfy high-level commands. The project was hugely influential in AI planning and spawned a number of noteworthy and to this day relevant techniques. Amongst them was the input language to the Stanford Research Institute Problem Solver (STRIPS) [FN71], a software system that could infer sequences of actions (i.e. plans) to solve complex problems. It also brought forth A* ([HNR68] and [HNR72]), a technique for best-first graph traversal derived from Dijkstra's algorithm [Dij59], that improved performance by incorporating heuristics and is still used in the majority of state-of-the-art planners today. Another important contribution attributed [LPW79] to the project was the *visibility graph* method for determining Euclidean shortest paths between points on a plane with obstacles.

Over the last 40 years, its progeny has made great strides. In its time it took Shakey's PDP-10 multiple hours to come up with and execute its plans of moving between positions and pushing boxes. In the 2007 Urban Challenge (see Figure 1.3), the winning autonomous vehicle managed to traverse a 96km urban area course in little less than 4 hours while obeying all traffic laws and avoiding other agents on the course [MC07], reasoning about a complex, dynamic environment in real time. Apart from increases in available processing power this

FIGURE 1.4 In the game "Empire" by Sega the AI uses goal oriented action planning to direct its units in real-time [Pav08].

was made possible by advancements in combinatorial optimization techniques and the development of more expressive planning frameworks which allow modeling (amongst others) multiple agents, dynamic domains, imperfect sensing, uncertain action effects and durations.

1.3 Classical Planning

The reasoning framework behind what in 1970 was described as "the first electronic person" [Dar70] is nowadays referred to as "classical planning". Perhaps unsurprisingly, it pertains to the most restricted class (in regards to environment phenomena) of planning problems. Classical planning assumes full observability, a finite (in terms of possible states and actions), static and deterministic environment as well as episodic progression of time (i.e. time passes in abstract episodes and steps, in each of which the agent chooses an appropriate action). World state is usually represented as conjunctions of (possibly negated) atomic formulae and agent actions as corresponding logical operators. Valid plans in this framework are action sequences that achieve the agent's goal if consecutively executed in its current state. The quality or merit of a valid plan in classical planning is generally defined as inversely proportional to its length. Hence, rational agent behavior comprises of finding and executing the shortest valid plan. A close relative of classical planning is automated theorem proving where the task is to deduce (find a plan) some theorem (goal) from a set of axioms (the initial state) through repeated application of deduction rules (the actions). In delineation, theorem proving is not an optimization problem as the task commonly involves no valuation on deductions (i.e., one is as good as any). More generally, classical planning is a discrete optimization problem, or more

specifically one of combinatorial optimization. Combinatorial optimization problems generally comprise of selecting some optimal object from a finite (or at least countable) set of candidates [Sch03] too large to be tackled by exhaustive exploration.

Many different approaches have been developed to solve classical planning problems over the last five decades. They can be roughly categorized into *search-based* and *logic-based* approaches. The former are also known as *explicit state* planners - they work by beginning from the initial state systematically applying the logical operators corresponding the agents actions to generate new reachable states (or apply their inverses from the goal state for *regressive-search* planners). In other words, they traverse (or search) the domain graph beginning from the initial state until they discover the shortest path to any goal state. Most such planners make use of a guidance function which encodes domain knowledge, a so called *heuristic*, to speed up the traversal. Notable examples from this class that significantly improved the state-of-the-art (mostly through more advanced heuristics) include the General Problem Solver (GPS) [NSS+59], the original STRIPS solver [FN71], UnPOP [McD96], the Fast Forward Planning System (FF) [HN01], the Fast Downward Planning System (FD) [Hel06a] and LAMA [RW10]. The latter class is more varied in their approaches. Most of its members attempt to constructively prove the existence of a solution. Notable examples are GRAPHPLAN [Blu95, BF97], which searches a derivative (of the original planning problem), small search space describing mutual interactions of agent operators and subgoals for plan candidates and tries to refine these candidates into plans for the original problem, the SATPlan family of planners [KS92, KS96, KS99, KSH06] that bijectively map classical planning problems to SAT instances which they then attempt to solve and lifted planning approaches such as GAMER [EK08a] where states are grouped in equivalent sets and successor sets generated in bulk. Common to these approaches is that at some point they all break down to traversals of implicitly defined graphs derived from the original problem description, i.e. problems of combinatorial search and optimization. In fact, newly developed approaches to classical planning have repeatedly stimulated new insights and advances for other approaches. This has been particularly the case for heuristic search-based approaches, where for example ideas such as GRAPHPLAN's exploration of the planning graph led to the development of a powerful, domain-independent class of heuristics (in this particular case, the h^m or *critical-path* family of heuristics [HG00]). For brevity and clarity, I restrict the following discussion of classical planning to (mostly) search-based approaches. I want to note however, that the main contributions of this thesis apply to combinatorial search in general and are hence *not* restricted to any particular approach.

Over the last 40 years classical planning has matured, been successfully applied to ever

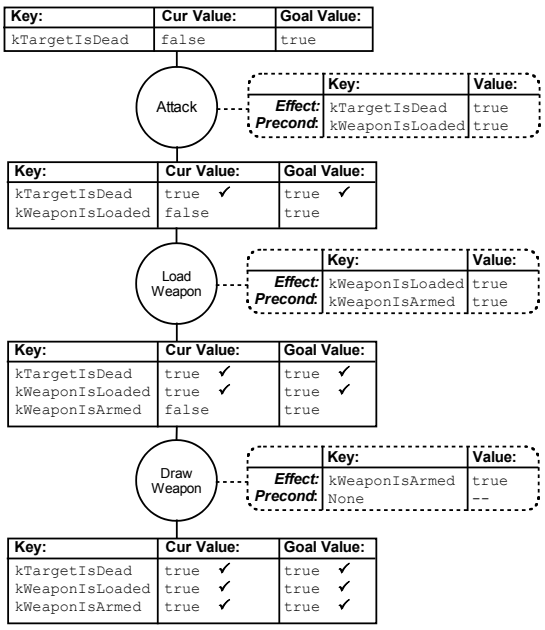


FIGURE 1.5 Example plan for an non-player-controlled character in the video game "F.E.A.R" generated through regression search (from [Ork03]).

larger problems and remained highly relevant. As a technology it has in recent years successfully cracked the threshold to commercial applications. Particularly in the field of entertainment software, planners are at work in many recent high-profile releases (for an example see Figure 1.4). The growing complexity of game-worlds make traditional approaches such as *finite state machines* (FSM) and *rule-based systems*, where the designer, in order to achieve consistently believable behaviors, must more or less anticipate the bulk of possible game situations and explicitly model corresponding behavior challenging. Planning eases this burden considerably by reasoning over game settings in real-time and stringing together appropriate sequences of atomic agent actions. The most prolific de-facto standard in the field is *goal oriented action planning* (GOAP) [Ork06], a simplified variant of STRIPS based on regression search.

Figure 1.5 shows a simple example plan for a non-player character comprising of atomic actions defined by FSMs. Efforts are currently underway (see [OBdBB+04] and [YdB06]) to create a standardized problem definition language for game AI in order to foster development of middleware solutions. Inevitably the underlying technology has recently spread to less diverting applications such as missile route planning and targeting [DS07] as well as threat analysis [Bja08]. Other commercial deployments include Training and Simulation systems [DEZGK11], traffic management for elevator systems [Koe01] and smart greenhouses

[HL10]. While not strictly classical planning, the fundamentals have been successfully transferred to query optimization for *relational database management systems*, where model-based planner are used to derive cost-optimal access plans for a given SQL[2] query and database [YL89]. In many other domains featuring combinatorial optimization problems such as airport ground traffic control [THN04] and gene-analysis [UE10], current domain-independent planners are on the verge of achieving a feasible level of maturity and performance. Being able to cast such problems in standardized descriptions (and solve them) alleviates the need to develop expansive special purpose solvers and is hence commoditizing them, a process that will allow organizations to reap the economical benefits of combinatorial optimizations in a wide variety of domains with little set-up costs.

1.4 State of the Art

Most of the significant leaps in optimal classical planning over the last decades stemmed on the one side from research into novel informed search algorithms that aimed to reduce the memory requirements of A* such as IDA* [Kor85], frontier search [KZTH05] and breadth-first heuristic search [ZH04a]. The other fruitful branch has been *heuristics*. In the context of classical planning, heuristics are essentially approximate guiding functions that (in the best case) lead the search algorithm towards promising solution candidates. For many domains, discoveries of suitable or improved heuristics have led to tremendous increases in the scope of problems planners can handle. The contributions are too extensive to list here but a selection of influential developments were the families of *relaxation* heuristics [BG01, HN01], *critical path* heuristics [HG00], abstraction heuristics [CS98] and landmark heuristics [Hel10, KD09].

1.5 Motivation

In recent years, research into classical planning has been driven along two major axes.

The first axis is domain-independent planning and the respective community is very much focused on the International Planning Competitions (IPC) held in the context of of the main conference in the field, the International Conference on Automated Planning and Scheduling (ICAPS). These competitions have led to the development of the ever evolving Planning Domain Definition Language (PDDL)[MGH+98, YL03, FL03], the de-facto standard in academia for the description of planning problems along with a wide variety of benchmark instances. The performance of different approaches in these competitions have traditionally

[2] Structured Query Language

strongly influenced the direction of the community. The winning entry in the classical planning track (for the detailed results, see [HDR08]) of the competition in the year 2008[3] was GAMER[EK08a], a symbolic heuristic search planner build around the idea of representing state-sets as binary decision diagrams[Bry86] (BDDs), a time and, often, very space efficient data structure. GAMER's approach is noteworthy as the planner also managed to win in the two other tracks it competed in (fully-observable, non-deterministic track [BB08] and the net-benefit optimization track [HDR08]). The best performing planner in the sequential optimal track however turned out to be a straightforward breadth-first search (BFS) provided as a reference implementation by the competition organizers. In these competitions, planners are tested on a variety of domains (priorly undisclosed to the participating teams), each of which sport multiple problem instances of increasing difficulty. A planner is awarded a point, if it manages to solve an instance in a fixed allotment of time. While the BFS could only solve relatively simple instances, it could do so across all domains, whereas the competitors generally either failed completely in a domain or predictably dominated the BFS. BDDs are a somewhat difficult member in the family of classical planning techniques. One of their caveats is that they result in very small representations for some domains, while more or less leading to exponential bloat in others, another is that they do not integrate easily with many algorithms and heuristics, a particular problems are strong heuristics that give very accurate cost predictions and result in the algorithms having to handle large numbers of individually small, value-equivalent state-sets. The resulting trade-off is generally not clear cut, for example the runner up planner in the classical planning track, HSP^*_F, an A* based heuristic search planner with such a strong heuristic (a variant of additive h^2 [HBG05]) came in barely behind GAMER (115 vs. 111 points).

The other axis of research is domain-dependent planning, where a common approach is to employ standard search algorithms in combination with strong domain-dependent heuristics and hand-crafted, efficient state representations. This combination often enables domain-dependent planners to outperform their more general peers by several orders of magnitude. Examples of such efficient encodings and heuristics can be found in [KZTH05], [KF07], [HR10]and [BK10]. The practical potential of efficient state-representation as well as the limits in both domain-applicability and combinability with heuristics and algorithms of current techniques were the primary motivation behind the development of the Level-ordered Edge Sequence, one of the main contributions of this thesis.

Planning is a computationally hard problem. Even classical planning is PSPACE-complete and NP-hard only with severe restrictions on the problem specification (c.f. section 2.1.3).

[3] IPC-2008, part of the *International Conference on Automated Planning and Scheduling* 2008

Parallel search algorithms have been around for many years [KR87, RK87, BK91] and intuitively seem like an obvious fit for planning. Despite of this, current state-of-the-art planners are exclusively based on sequential implementations. The reason is due to the large utility gain planners enjoy from exploiting dynamic programming (c.f. the duplicate detection example for the 15-puzzle above). The price for employing dynamic-programming is that search algorithms have to manage a large amount of state during execution. So much of it in fact, that memory use is generally the primary bottleneck in planners and much work has been done on efficiently employing external memory in search (see [ZH04b, Kor04, ZH07a] amongst others). Recent parallel dynamic-programming algorithms (e.g. [EHMN95, BLZR09, KFB09]) combine the benefits of a vastly reduced search space with better utilization of existing computational resources, but result in a higher memory-footprint per encountered state - seldom an acceptable trade-off in practice. Another significant hurdle in parallelizing planning is that this search state changes at a very high frequency and generally needs to be kept consistent across all cooperating processes to guarantee the soundness of the respective algorithm. The resulting synchronization overhead is often so significant, that the parallel algorithm barely (if at all) outperforms its sequential counterpart, even when running on a large number of processors. Burns et Al. [BLZR09] provide a good overview on the relative performance of current, parallel best-first algorithms on a number of standard problems. Yet, as standard workstations sport more and more processors and cloud computing enable the economical use of large clusters, the potential benefits of parallel planning are increasing significantly. This motivated the development of Parallel Edge Partitioning (PEP) the second main contribution of this thesis, a very low overhead, lock-free synchronization scheme that exploits the innate topology of planning problems and integrates straightforwardly into a wide range of best-first search algorithms.

Both LOES and PEP are designed to integrate as orthogonally as possible into the heuristic search toolkit. LOES' aims to increase the space efficiency of search algorithms and memoization heuristics as well as the effective I/O bandwidth of external memory and distributed planning systems by simply replacing standard state-set representations with LOES based ones, thereby alleviating the primary bottleneck of classical planning. The analogous more or less holds for the parallelization of best-first search algorithms based on PEP. PEP straightforwardly allows to partition layer expansions into sets of distinct independent jobs that can be executed in parallel without the further synchronization. It enables the parallel adaptation of existing search algorithms and allows best-first search to exploit the nowadays prevalent concurrent computing platforms without impacting spatial efficiency.

1.6 Contributions

The contributions of this thesis can be categorized into three areas. The two main contributions are general techniques for state-set representation in classical planning and parallelization of the underlying combinatorial search algorithms. The third contribution is in the description of a heuristic and search algorithm for target-value search, a class of combinatorial search problems that occurs in system diagnosis. In detail they are:

LEVEL-ORDERED EDGE SEQUENCE An adaptive succinct data structure for memoization [Mic68] in combinatorial search. This research was motivated by the results of international planning competition 2008. *Gamer* [EK08a], the winning entry in the sequential optimal track showed the potential of space-efficient state-set representations for classical planning. The outcome is notable for the fact that gamer uses symbolic (or lifted) search, an approach that for classical planning is usually outperformed by state-of-the-art explicit-state search algorithms. Its representation is based on binary decision diagrams (BDDs) [Bry86], a technique that in recent years gained considerable traction in planning but sports a number of notable drawbacks. First is its lack of robustness. The search domain needs to exhibit a suitable structure for the method to be space efficient. If the domain lacks such structure (and many do), BDDs can quickly outgrow standard set representations by orders of magnitude. This pattern is apparent in the competition results (see [HDR08]) - where gamer "works" it usually dominates all competitors, where it does not, it fails to solve even the easiest instances. Second, operations pertaining to individual states are relatively costly in BDDs, making it a hard to integrate with many search techniques and heuristics (hence gamer using symbolic search). Lastly, BDDs only offer very limited possibilities to associate ancillary data with individual states without incurring a large hit to its space efficiency. The last issue is of general importance for domain independent planning. Many of the strong and general heuristics developed so far cannot feasibly be integrated into domain independent solvers. Abstraction heuristics in particular often show very good performance when coupled with efficient, hand-crafted (domain dependent) representations, but are limited to very coarse abstractions with current general representations. Other heuristics need to be expensively computed from scratch for each generated state, relegating them to few domains where their computational cost is outweighed by their performance. Suitable representation techniques could ease this burden by enabling memoization techniques such as dynamic programming, where solutions to common subproblems are retained for future use. The Level-Ordered Edge Sequence (LOES) is a space-efficient data structure for state-sets and maps represented

as prefix trees that allows efficient member queries, enumerations and serves as a *minimal perfect-hashing* function for members. Underlying the data structure is a pointerless level-order tree encoding, similar to the static data-structures used to encode large suffix trees. LOES adapts these principles to the dynamic environment of dynamic programming allowing amortized $O(\log(n))$ insertions and worst-case $O(\log(n))$ lookups and value updates. Its principles and use in search algorithms are partly discussed in [SZ11b]. Its suitability as a representation for abstraction heuristics in [SZ11a].

PARALLEL EDGE PARTITIONING In order to continue to benefit from advances in semiconductor technology, search algorithms must be adapted to exploit hardware parallelism. Unfortunately most state-of-the-art heuristic search algorithms cannot be trivially parallelized. The central culprit is duplicate detection, i.e. keeping track of what states have already been visited during the search. Modern classical planners (depending on the domain) generate on the order of a million states per second per processor, all of which have to be tested against (and potentially update) the set of previously generated states. At such query and update rates, straightforward adaptations based on OS synchronization primitives suffer from so much overhead that they often barely outperform their single-threaded counterparts on massively parallel machines. Omitting or delaying duplicate detection is only feasible in few domains. As part of PARC's parallel planning effort, we developed Parallel Edge Partitioning (PEP), a divide and conquer approach for best-first search that exploits problem structure to parallelize graph search with low synchronization overhead. The idea is to automatically generate an abstraction of the search domain, partition all shared state-sets according to this abstraction and leverage said compartmentalization to avoid synchronization overhead for duplicate detection. PEP is an evolution of a prior approach developed at PARC and the Mississippi State University [ZH07b] characterized by a simpler and (computationally) cheaper synchronization model and most importantly a dependency *only on explicitly expressed* problem structure (in the domain description). These properties allow it to be combined with many state-of-the-art best-first search algorithms and to be used in basically any classical planning domain. This is joint work with Rong Zhou and a first outline of the technique was published in [ZSH+10].

COMBINING GUIDED AND UNGUIDED SEARCH FOR TARGET-VALUE-PATH PROBLEMS A central part of PARC's *Pervasive Diagnosis* [KPD+10] framework is a planner for the generation of informative production plans. The planner has to solve a class of combinatorial optimization problems with a non-additive cost functions which we coined

target-value path problems. In these problems, the planner has to find the (set of) path(s) between to given nodes in a graph with value closest to some given target-value. In non-cyclical graphs, such problems can be decomposed into a part with an additive cost-problem solvable by standard best-first search algorithms and non-additive part that has to be tackled with unguided search through a novel class of interval heuristics. Contributions here include a novel subclass of abstraction heuristics termed *multi-interval heuristics*, a dynamic-programming algorithm for their construction and a depth-first search algorithm that exploits this structure by combining guided and unguided search. This work was first published in [SKP+09].

CHAPTER 2

Preliminaries

In this chapter, I will give an overview of the fundamental building blocks of a classical planner. At the beginning is an overview of *propositional STRIPS*, a widespread formalism for classical planning.This is followed by an example problem, its STRIPS representation and its interpretation as a *graph search problem*. From there I will discuss basic search algorithms and end with the concept of *heuristics* and their incorporation into the graph search.

2.1 Classical Planning

One of the oldest planning formalisms is the input language of the Stanford Research Institute Problem Solver or STRIPS, a planner originally designed for the Shakey project, which in its relevance has so far outstripped the planner it was developed for that the acronym nowadays generally denotes the language. While STRIPS itself is still in widespread use today, it also forms the base of most other current classical planning formalisms such as the *Planning Domain Definition Language* (PDDL) [MGH+98], GOAP[Ork05] and *Simplified Action Structures* (SAS and SAS+) [Kle90].

2.1.1 Propositional STRIPS

In propositional planning, world states and goals are modeled as logical sentences and operators as inference rules. Formally, a propositional planning instance is a 4-tuple $\langle P, O, i, G \rangle$ with components (see [Byl94]):

- P, a set of propositional variables (or conditions).
- O, a set of operators or inference rules, where each is given as a 4-tuple of *subsets* of P of the form $\langle P_{true}, P_{false}, Q_{true}, Q_{false} \rangle$ such that:

  - $P_{true} \subseteq P$, the variables of P that must be true for the operator to be executable (i.e. the *positive preconditions*).
  - $P_{false} \subseteq P$, the variables of P that must be false for the operator to be executable (i.e. the *negative preconditions*).
  - $Q_{true} \subseteq P$, the variables of P that are made true by the operator (i.e. the *positive postconditions*).
  - $Q_{false} \subseteq P$, the variables of P that are made false by the operator (i.e. the *negative postconditions*).
  - $P_{true} \cap P_{false} = \emptyset$ and $Q_{true} \cap Q_{false} = \emptyset$
- $i \subseteq P$, the initial state given as the subset of variables of P that are true.
- G, the goal given as a 2-tuple of subsets of P of the form $\langle G_{true}, G_{false} \rangle$
  - $G_{true} \subseteq P$, the variables of P that must be true.
  - $G_{false} \subseteq P$, the variables of P that must be false.
  - $G_{true} \cap G_{false} = \emptyset$

A state is an assignment to all propositional variables in P. In *propositional* STRIPS (pSTRIPS)[1] states are represented as subsets s of P with the interpretation that all elements in s are assigned the value $true$, while all variables in $P \setminus s$ are $false$. State transitions for a STRIPS instance $\langle P, O, i, G \rangle$ are given by the transition function $\delta : 2^P \times O \to 2^P$ which is formally defined as

$$\delta(s, \langle P_{true}, P_{false}, Q_{true}, Q_{false} \rangle) = \begin{cases} s \setminus Q_{false} \cup Q_{true} & \text{if } P_{true} \subseteq s \wedge P_{false} \cap s = \emptyset \\ s & \text{else} \end{cases}$$

For convenience it is common to define an extension of δ for operator sequences as follows.

$$\delta(s, []) = s$$
$$\delta(s, [o_1, o_2, \ldots, o_n]) = \delta(\delta(s, o_1), [o_2, \ldots, o_n])$$

Another convenient short-hand is to define the successors of a state $\delta(s)$.

$$\delta(s) = \{s' \in 2^P | \exists o \in O \quad s' = \delta(o, s) \wedge s' \neq s\}$$

[1] the original STRIPS as given by [FN71] only allows *positive* preconditions and goal conditions, resulting in a slightly reduced expressiveness [Bae95]

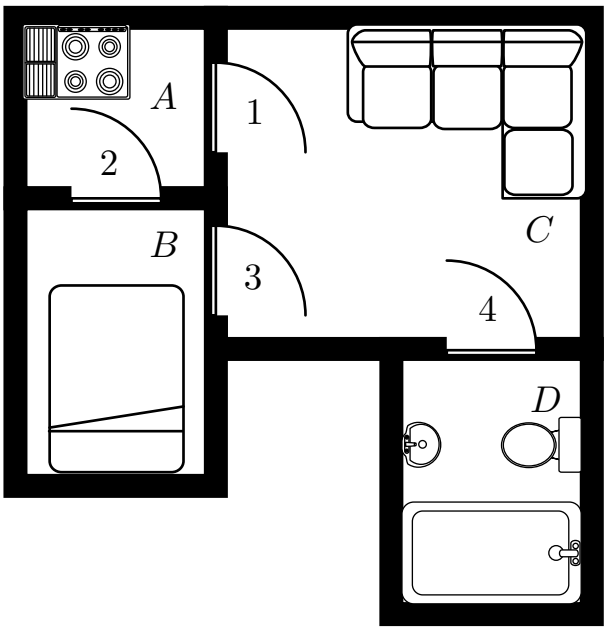


FIGURE 2.1 The apartment domain comprising of four rooms $A \ldots D$ and four doors $1 \ldots 4$.

An s-plan ($s \subseteq 2^P$) for a pSTRIPS instance $\langle P, O, i, G \rangle$ is an operator sequence $\pi_s = [o_1, \ldots, o_n]$ ($o_i \in O$), such that $G_{true} \subseteq s'$ and $G_{false} \cap s' = \emptyset$ with $s' = \delta(s, \pi_s)$, or informally a sequence of operators that when executed in s results in a successor state s' that fulfills the instance's goal conditions. An i-plan is a plan for the initial state. An *optimal* plan π_i^* is the shortest such sequence of operators.

The pSTRIPS formalism is hence a very simple logical framework which allows to model an agent's context as a set of variables and his or her possible actions within this context as corresponding operators manipulating said variables. Both sets together are referred to as the *domain* of a planning problem. Within this domain, jobs for the agent can be defined by describing both current and desirable conditions in the form of assignments to the propositional variables. Such tasks are called *problem instances* in planning terminology.

2.1.2 The Apartment Domain

To provide some grounding, I want to consider a simple example domain of navigating a small apartment (see Figure 2.1). The (somewhat closed off) apartment comprises of a kitchen (A), a bedroom (B), a living room (C) and a bathroom (D) with four connecting doors ($1 \ldots 4$). To navigate it, an agent can open and close doors (if it is in the same room) and move between adjacent rooms. A corresponding propositional STRIPS formalization with 8 propositional

variables and 24 operators can be given as follows:

$$P = \left\{ \begin{array}{ll} p_A, & \text{“agent is in room } A\text{”} \\ p_B, & \text{“agent is in room } B\text{”} \\ p_C, & \text{“agent is in room } C\text{”} \\ p_D, & \text{“agent is in room } D\text{”} \\ p_1, & \text{“door 1 is open”} \\ p_2, & \text{“door 2 is open”} \\ p_3, & \text{“door 3 is open”} \\ p_4 & \text{“door 4 is open”} \end{array} \right\}$$

$$O = \left\{ \begin{array}{ll} o^1_{A\to C} = \langle \{p_1, p_A\}, \emptyset, \{p_C\}, \{p_A\} \rangle, & \text{“move from room } A \text{ to room } C\text{”} \\ o^1_{C\to A} = \langle \{p_1, p_C\}, \emptyset, \{p_A\}, \{p_C\} \rangle, & \text{“move from room } C \text{ to room } A\text{”} \\ o^2_{A\to B} = \langle \{p_2, p_A\}, \emptyset, \{p_B\}, \{p_A\} \rangle, & \text{“move from room } A \text{ to room } B\text{”} \\ o^2_{B\to A} = \langle \{p_2, p_B\}, \emptyset, \{p_A\}, \{p_B\} \rangle, & \text{“move from room } B \text{ to room } A\text{”} \\ o^3_{B\to C} = \langle \{p_3, p_B\}, \emptyset, \{p_C\}, \{p_B\} \rangle, & \text{“move from room } B \text{ to room } C\text{”} \\ o^3_{C\to B} = \langle \{p_3, p_C\}, \emptyset, \{p_B\}, \{p_C\} \rangle, & \text{“move from room } C \text{ to room } B\text{”} \\ o^4_{C\to D} = \langle \{p_4, p_C\}, \emptyset, \{p_D\}, \{p_C\} \rangle, & \text{“move from room } C \text{ to room } D\text{”} \\ o^4_{D\to C} = \langle \{p_4, p_D\}, \emptyset, \{p_C\}, \{p_D\} \rangle, & \text{“move from room } D \text{ to room } C\text{”} \\ o^1_{\text{open from } A} = \langle \{p_A\}, \{p_1\}, \{p_1\}, \emptyset \rangle, & \text{“open door 1 from room } A\text{”} \\ o^1_{\text{open from } C} = \langle \{p_C\}, \{p_1\}, \{p_1\}, \emptyset \rangle, & \text{“open door 1 from room } C\text{”} \\ o^1_{\text{close from } A} = \langle \{p_A, p_1\}, \emptyset, \emptyset, \{p_1\} \rangle, & \text{“close door 1 from room } A\text{”} \\ o^1_{\text{close from } C} = \langle \{p_C, p_1\}, \emptyset, \emptyset, \{p_1\} \rangle, & \text{“close door 1 from room } C\text{”} \\ o^2_{\text{open from } A} = \langle \{p_A\}, \{p_2\}, \{p_2\}, \emptyset \rangle, & \text{“open door 2 from room } A\text{”} \\ o^2_{\text{open from } B} = \langle \{p_B\}, \{p_2\}, \{p_2\}, \emptyset \rangle, & \text{“open door 2 from room } B\text{”} \\ o^2_{\text{close from } A} = \langle \{p_A, p_2\}, \emptyset, \emptyset, \{p_2\} \rangle, & \text{“close door 2 from room } A\text{”} \\ o^2_{\text{close from } B} = \langle \{p_B, p_2\}, \emptyset, \emptyset, \{p_2\} \rangle, & \text{“close door 2 from room } B\text{”} \\ o^3_{\text{open from } B} = \langle \{p_B\}, \{p_3\}, \{p_3\}, \emptyset \rangle, & \text{“open door 3 from room } B\text{”} \\ o^3_{\text{open from } C} = \langle \{p_C\}, \{p_3\}, \{p_3\}, \emptyset \rangle, & \text{“open door 3 from room } C\text{”} \\ o^3_{\text{close from } B} = \langle \{p_B, p_3\}, \emptyset, \emptyset, \{p_3\} \rangle, & \text{“close door 3 from room } B\text{”} \\ o^3_{\text{close from } C} = \langle \{p_C, p_3\}, \emptyset, \emptyset, \{p_3\} \rangle, & \text{“close door 3 from room } C\text{”} \\ o^4_{\text{open from } D} = \langle \{p_D\}, \{p_4\}, \{p_4\}, \emptyset \rangle, & \text{“open door 4 from room } D\text{”} \\ o^4_{\text{open from } C} = \langle \{p_C\}, \{p_4\}, \{p_4\}, \emptyset \rangle, & \text{“open door 4 from room } C\text{”} \\ o^4_{\text{close from } D} = \langle \{p_D, p_4\}, \emptyset, \emptyset, \{p_4\} \rangle, & \text{“close door 4 from room } D\text{”} \\ o^4_{\text{close from } C} = \langle \{p_C, p_4\}, \emptyset, \emptyset, \{p_4\} \rangle & \text{“close door 4 from room } C\text{”} \end{array} \right\}$$

The operators model conditions for and consequences of agent actions in the domain. Operator $o^3_{C\to B}$ for example is enabled, if the agent is in room C and door 3 is open. Upon execution, p_C is set to $false$ and p_B to $true$, denoting the agent's new whereabouts. Similarly $o^4_{\text{open from } D}$ is enabled if the agent is in the bathroom (p_D) with a closed door 4 ($\neg p_4$). Upon execution, p_4 is set to $true$.

One can now formulate tasks over this domains by giving initial assignments for all variables and goal assignments for some *non-empty* subset. A task such as “starting in the bathroom, with all doors closed, move to the kitchen” is then formalized as

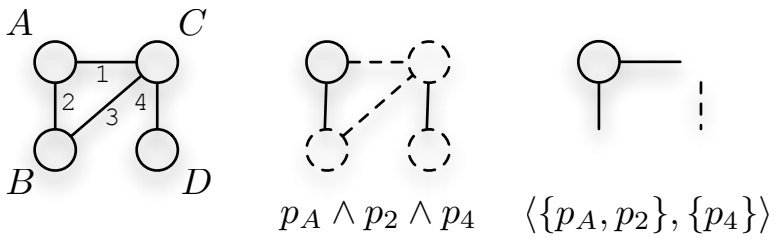


FIGURE 2.2 Graphical notation for apartment domain states. (left) The state graph with corresponding domain elements. (middle) Assignments *true* and *false* are represented by solid and dashed outlines respectively. (right) Goals simply omit elements for unqualified (or do not care) Variables.

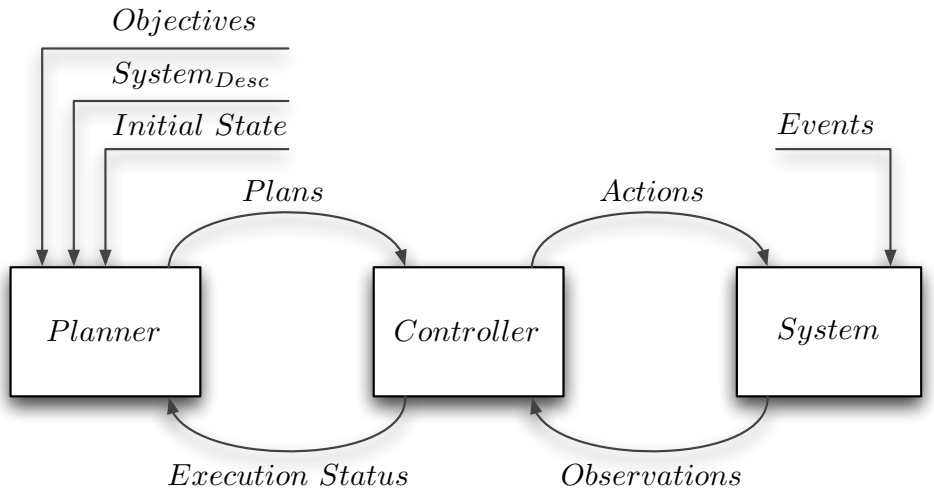


FIGURE 2.3 Abstract architectural view of a planning system.

$$i = \{p_D\}$$
$$G = \langle\{p_A\}, \emptyset\rangle$$

As the apartment domain will be used as an example throughout the chapter and the STRIPS syntax is somewhat abstruse, I will (where appropriate) use an equivalent graphical notation to denote states of the domain. It is informally introduced in Figure 2.2.

2.1.3 Classical Planning - Assumptions, Classification and Complexity

Over the last decades, a multitude of planning frameworks have been developed in the field of AI. Most of them were explicitly constructed for specific problems. According to their

domain properties, such as imperfect sensing, uncertain action outcomes, parallel actions, multiple agents, inherent system dynamics, varying action durations and concurrency, these frameworks differ widely in their expressivity. Figure 2.3 shows the architecture of a generic planning system. Given a description of the system, its (more or less known) initial state and the agent's objectives for the system, the planner computes a plan. This plan defines behaviors conditioned on the system state and is usually interpreted by a control layer. The control layer aggregates system observations into a (more or less complete) system state and dependent on that state derives appropriate actions from the plan. It reports its execution status to the planner which can, under certain conditions, lead to the computation of an updated plan. In addition to the agent's actions, the system is usually also influenced both by exogenous events (e.g. by other agents acting upon the same system or the system's internal dynamics). A common way of classification for planning problems is through dichotomies given by the following set of restrictive assumptions:

A1 - FINITE DOMAIN The domain comprises of only *finitely* many states, actions and events. The apartment domain sports 64 (meaningful) states and 24 actions. In an expanded domain with path-planning aspects, the position of the agents could be modeled with real-valued coordinates.

A2 - FULLY OBSERVABLE SYSTEM The controller always knows the *complete* state of the system. In the apartment domain, the position of the agent as well the state of all doors is known at all times. In a more complex setting, an agent in the bathroom might not be able to determine whether doors 1, 2 and 3 are open or closed without first leaving the bathroom.

A3 - DETERMINISTIC ACTIONS All actions have a *single, well-defined* outcome. E.g. (if enabled,) actions such as "move from A to B" or "close door 2" will always succeed in the apartment domain. In a more complex setting, an agent's actuators may have a chance of failing at these tasks (e.g. a robot's gripper might slip from the door handle).

A4 - STATIC SYSTEM The system is *not* subject to exogenous events (i.e. all changes to the system are due to the controller's actions). In a more complex setting, doors could be opened or closed due to draft or concurrently acting agents.

A5 - ATTAINMENT GOALS The planner's objectives take the form of a set of goal states G. A more complex domain might require defining action-costs conditioned on system state and tasking the planner with minimizing the accumulation of said costs over the agent's lifetime.

A6 - SEQUENTIAL PLANS Plans are *linearly ordered* sequences of actions. In more complex domains, agents might be capable of parallel execution of actions and/or actions might depend on the outcome of prior actions and system events.

A7 - IMPLICIT TIME Actions (and corresponding state transitions) happen *instantaneously* (i.e. there is no notion of action duration). A more complex model of the apartment domain might include distinct durations for all actions where furthermore the effects of an action in the domain could unfold gradually. As an example the agent might unlock and push a door, which now has enough momentum to open after some amount of time.

A8 - OFFLINE PLANNING The planner exists in an *open-loop-system* relationship with the controller, i.e. there is no feedback from the control layer. In a more complex domain, the underlying system description might be updated during plan execution necessitating plan adaptations (e.g. certain actuators of an agent could malfunction, doors might jam, etc.).

Frameworks such as propositional STRIPS which fulfill all these restrictive assumptions are classified as classical planning systems. While theoretically the "easiest" class of planning problems considered in AI they are in general still extremely computationally challenging. Deciding the existence of an i-plan[2] (the $PLANSAT$ or $PLANEX$ problem) for a propositional STRIPS instance is only polynomially bounded with severe restrictions on the operators and/or goals. In general the problem is $PSPACE$-complete[Byl91]. For a detailed overview of restrictions and corresponding complexity classes see figure 2.4. Erol et. al. showed [ENS92] that optimal propositional planning ($PLANMIN$) is NP-complete, if *no* negative postconditions or delete lists are allowed (i.e. $Q_{false} = \emptyset$) and $PSPACE$-complete otherwise and Bylander [Byl94] later refined these results to what is shown in figure 2.5.

These results serve well to frame the expectations of what a domain-independent planner can achieve. The additional restrictions one needs to impose to guarantee polynomial complexity in the (already very restricted) classical planning framework are too severe for most planning problems. For an argument against and in favor of such tradeoffs of tractability and expressiveness see [LB87] and [DW91] respectively. I will here not dwell further on this issue.

However it has been shown that many practical problems can be formulated in propositional planning respecting the restrictions for NP-complete boundedness. Earlier work concentrated on individual domains like the n-puzzles [RW90] or blocksworld [GN91] while recently this

[2]Such tasks, where the goal is to find some i-plan in absence of any optimality criterion is commonly referred to as *satisficing planning* in AI.

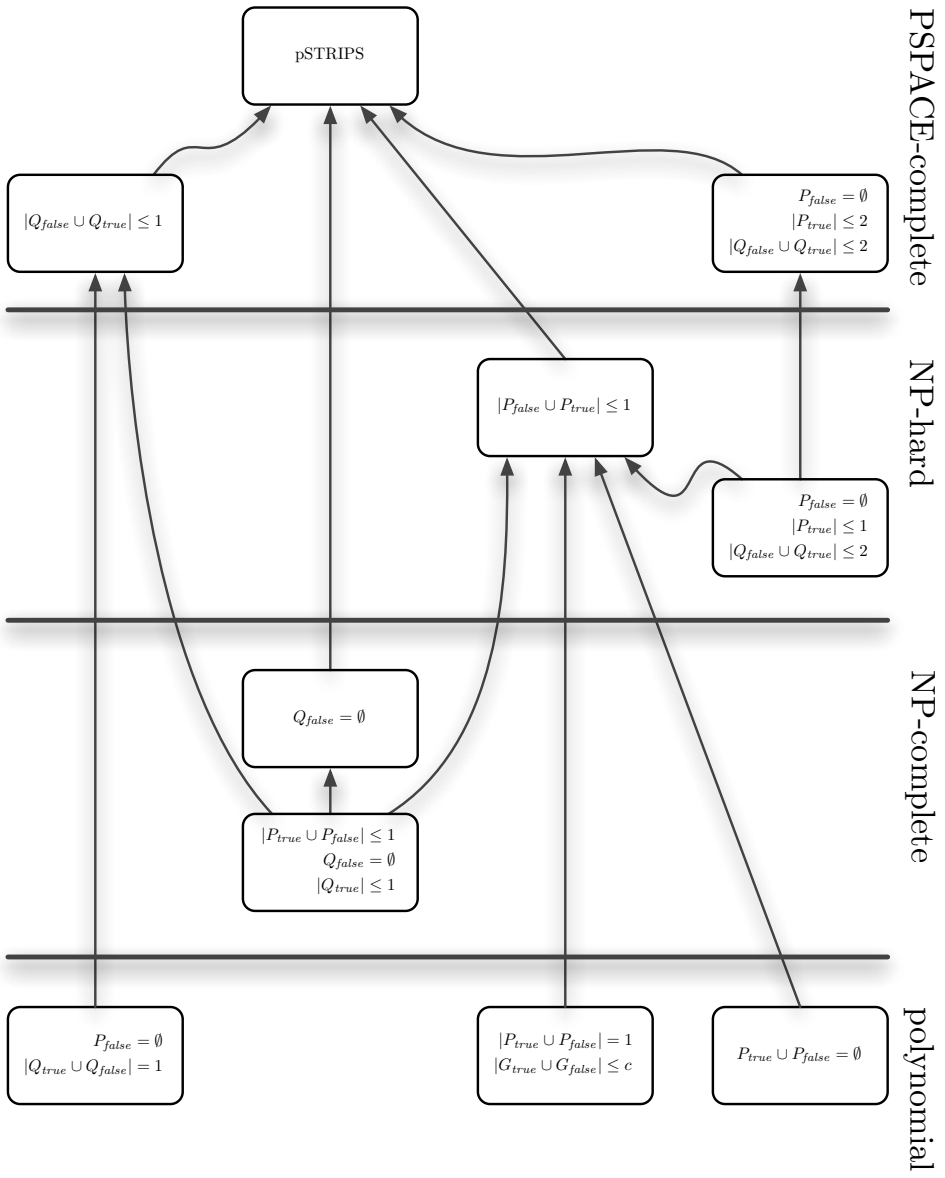


FIGURE 2.4 Computational complexity of satisficing planning in propositional STRIPS ($PLANSAT$) for different restrictions on the operator and goal sets after Tom Bylander (see [Byl91] for proofs).

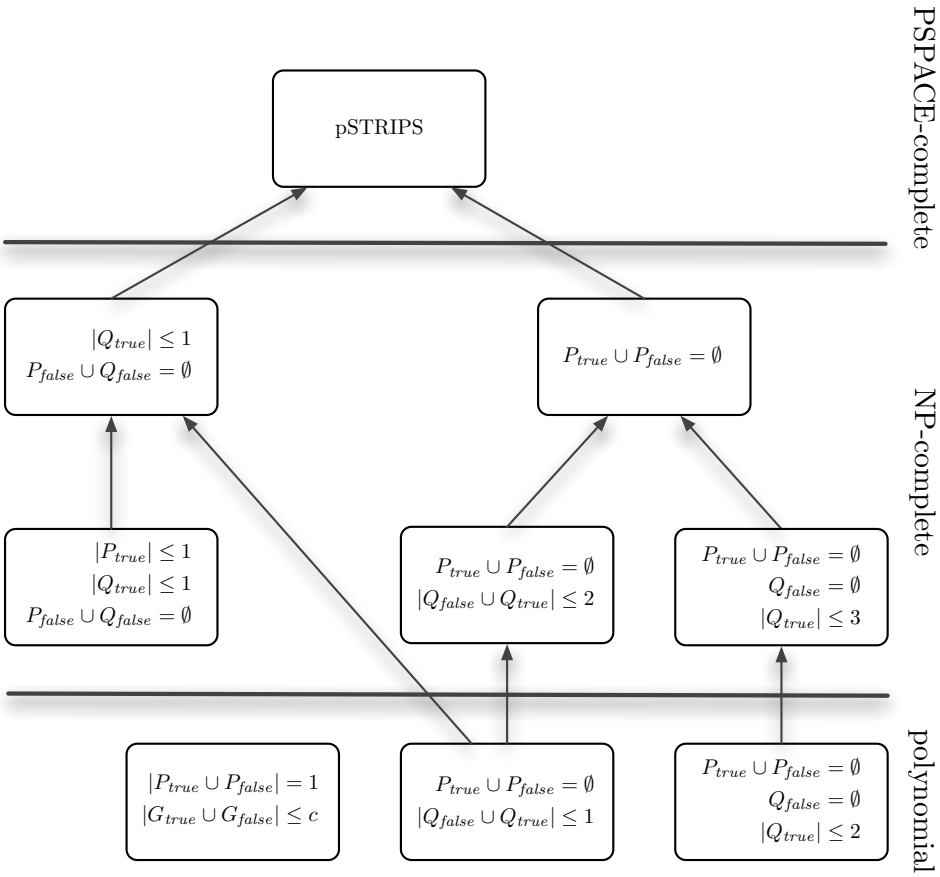


FIGURE 2.5 Computational complexity of optimal planning in propositional STRIPS ($PLANMIN$) for different restrictions on the operator and goal sets after Tom Bylander and Erol et. Al. (see [ENS92] and [Byl94] for proofs).

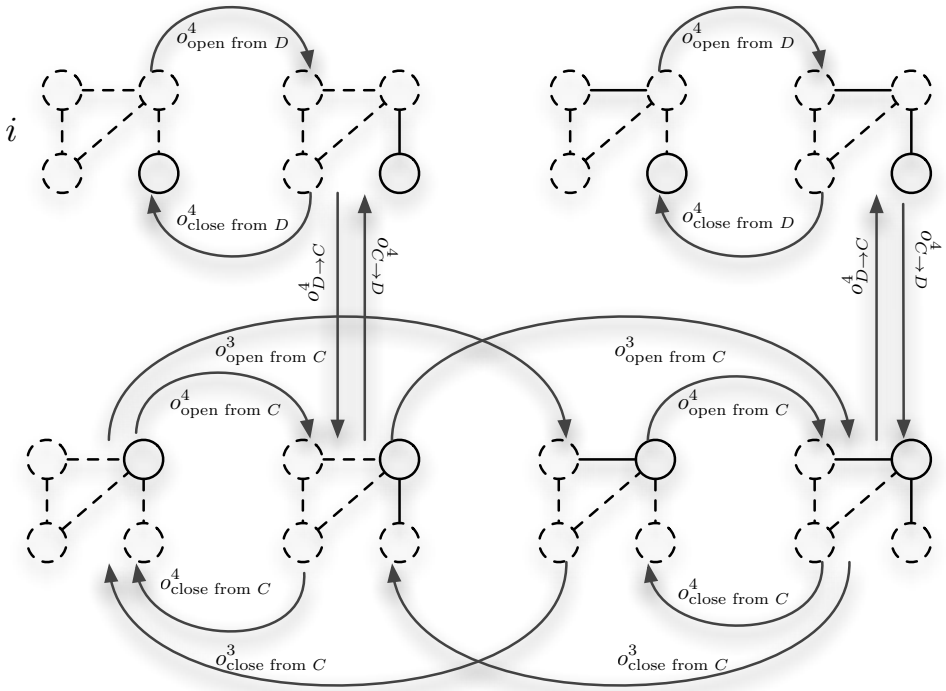


FIGURE 2.6 Subset of the apartment domain-graph pertaining to operating doors 3 and 4 and moving between C and D for an agent starting in D with all doors closed (state i, top-left). The complete graph (with a single agent) comprises of 64 states.

has been shown for entire classes of problems (see [Hel03] and [Hel06b]). Research into NP-complete problems(i.e. [CKT91],[MJPL92] and [MSL92]) suggests that they are generally hard only for a small share of their instances. Because of this, efforts to construct (feasible) domain-independent solvers have so far mostly focused on classical planning. Note that for the following, I will use STRIPS and pSTRIPS interchangeably.

2.2 Classical planning as a problem of combinatorial optimization

Having discussed the problem formulation, I will now tend to the actual planning, i.e. the task of deriving a plan from a STRIPS instance. Every STRIPS domain can be straightforwardly interpreted as a finite labeled graph, where for each triple $s, s' \in 2^P, o \in O$ with $\delta(s, o) = s'$ there is a directed transition from s to s' labeled o. Inert transitions or "no-ops" (i.e. where

$s = s'$) are generally omitted. Figure 2.6 shows part of the domain graph for an apartment instance where the agent starts in the bathroom with all doors closed.

In this graph the task is to look for a path from i, the initial state to some state $g \in G$. A Problem, where the goal is to look for some specific substructure of some given *discrete* structure is termed a *combinatorial search*. Problems, which in addition demand the minimization (or maximization) of some property of qualifying substructures are commonly classified as *combinatorial optimizations* [Knu73]. Hence satisficing classical planning (i.e. where one looks for some i-plan) lies in the former class and optimal classical planning (i.e. where one looks for the *shortest* i-plan) lies in the latter. For the remainder of this work, I will concentrate on optimal (classical) planning.

2.3 Graph-traversal algorithms

In order to find a suitable plan, the domain graph must be traversed (i.e. searched) in a systematic way. In the following, I will give a short introduction into the cardinal approaches to graph search.

2.3.1 Depth-first search

Algorithm 1: DBDFS

Depth-bounded depth-first search (recursive implementation);

Input: π a sequence of operators
Input: d the search depth limit
Output: an i-plan or $\perp$

```
if d > 0 then
    s ← δ(π, i);
    if s ∈ G then
        return π;
    end
    foreach o ∈ O s.th. δ(s, o) ≠ s do
        DBDFS(π ∘ o, d − 1);
    end
end
```

The idea of *depth-first search* (DFS) is that starting from some initial state, one explores as far as possible along each path. Once no further exploration is possible in the current path

(i.e. the last state has no successors), one steps back to the most recent state with unexplored successors and continues from there. In other words, DFS is a straightforward *backtracking* technique [CLR90].

As domain graphs are in general *cyclical*, basic DFS is *not complete*. In the apartment domain, every (feasible) state has at least one successor, so the procedure would never backtrack and indefinitely append to the initial path. A slight modification - a *depth-bound*, i.e. some maximal allowable path-length, which enforces backtracking upon transgression - can cure this. Furthermore, exploration stops once the current path is an i-plan, or in other words the last state of the path is a goal state. See Listing 1 for the pseudo-code.

Algorithm 2: IDDFS

Iterative deepening depth-first search;

Output: an optimal i-plan or $\perp$

$\pi \leftarrow \perp$;
$d \leftarrow 1$;

while $\pi = \perp$ **do**
  $\pi \leftarrow$ DFS-DL($[], d$);
  $d \leftarrow d + 1$;
end
return π ;

The idea then is to run multiple DFS from i with increasing depth limits (beginning with one) until a solution is found. This procedure is *optimal* (i.e. the i-plan it computes is *at least as short* as any other plan for the problem) and *complete* (i.e. if there is an i-plan, it will eventually be found). This algorithm was first described in [SA77] and is today known as *iterative deepening depth-first search* (IDDFS)[RNC+10] and given in Listing 2.

Figure 2.7 gives an example apartment instance and shows the candidates generated by IDDFS during its graph traversal. Three runs of DBDFS are necessary to find an optimal i-plan. The number of candidates for a run is usually *exponential* in its depth-limit, hence subsequent runs regenerating candidates of previous runs results in only little relative overhead.

2.3.1.1 Shortcomings

Of note are candidates 4 and 8 in the last run. These plans, when executed in i result in i. They stem from cycles in the domain-graph. Note that successor states in classical planning only depend on the current state and not on the sequence of actions that lead to the state. Stochastic processes with this characteristic are said to exhibit the *Markov* property, but the

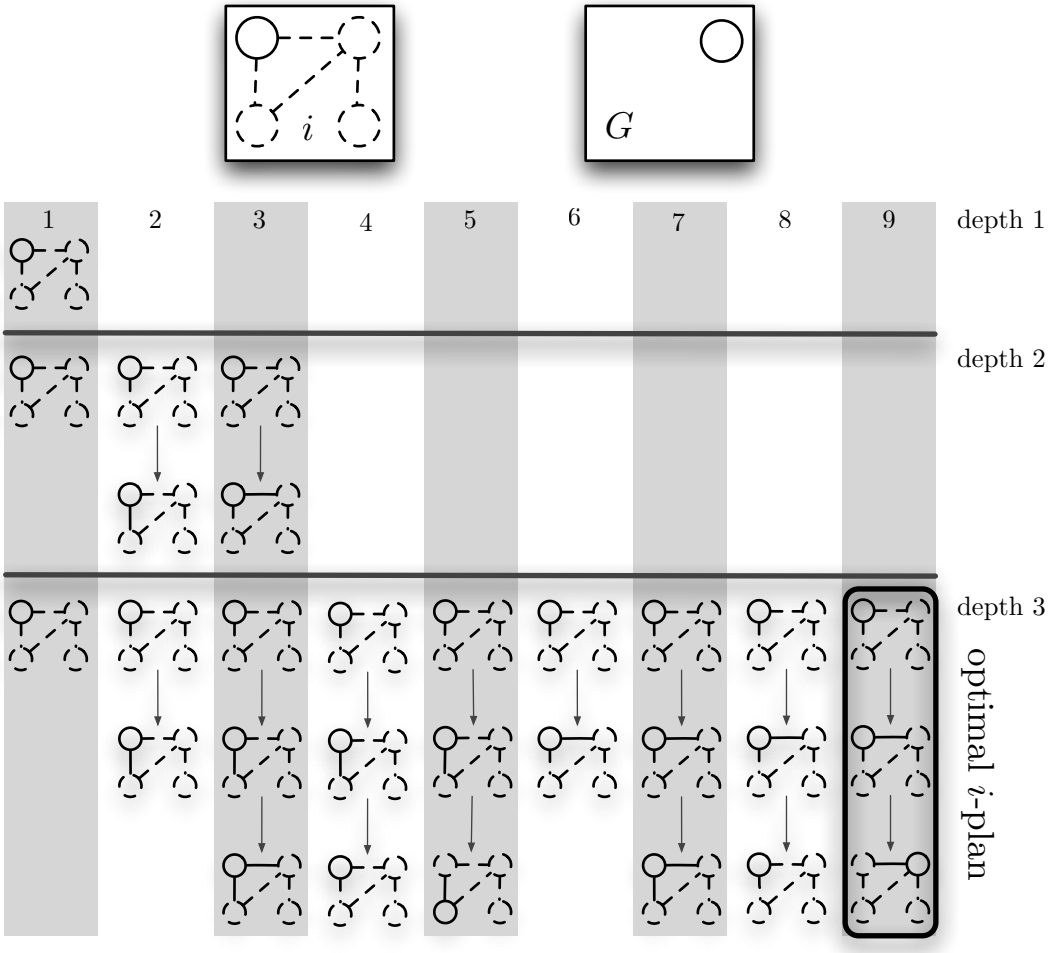


FIGURE 2.7 Candidates (worst-case) as generated by IDDFS for an apartment instance where the agent starts in the kitchen with all doors closed and wants to move to the living-room.

term is sometimes used in the context of deterministic problems as well. When the goal is to find the shortest plan, following cycles in the domain graph is *not* advisable. Intuitively this is easy to see - theoretically it stems from optimal classical planning problems exhibiting *optimal substructure*. This property informally states that every part of a (structured) optimal solution of a problem is in itself an optimal solution to to one of its subproblems (for a formal definition see [CLR90]). Concretely given a pSTRIPS instance $S = \langle P, O, i, G\rangle$ with some optimal i-plan $[o_1, \ldots, o_n]$ it holds that for any pair of operators o_i and o_j (w.l.o.g. $1 \leq i < j \leq n$) and corresponding states $s_{i-1} = \delta([o_1, \ldots, o_{i-1}], i)$ and $s_j = \delta([o_1, \ldots, o_j], i)$, $[o_i, \ldots, o_j]$ is an optimal i-plan for the related instance (a subproblem) $S' = \langle P, O, s_{i-1}, \langle s_j, P \setminus s_j\rangle\rangle$. The proof by contradiction is straightforward - assume $[o'_1, \ldots, o'_m]$ is a shorter i-plan than $[o_i, \ldots, o_j]$ for S'. Then $[o_1, \ldots o_{i-1}, o'_1, \ldots, o'_m, o_{j+1}, \ldots, o_n]$ is an i-plan for S (because of the markov property) and shorter than $[o_1, \ldots, o_n]$. Hence $[o_1, \ldots, o_n]$ cannot have been an optimal i-plan for S to begin with.

A related form of redundancy can be seen with candidates 3 and 7. While both stem from applying different plans at i, they result in the same state. Due to the optimal substructure of the problem, any optimal i-plan of the original instance "passing through" this state, will have an optimal i-plan for the corresponding subproblem as a *prefix*. Furthermore from the markov property follows that *all* optimal i-plans of the subproblem are interchangeable. Hence all plans that lead from i to a state s form an *equivalence class* and can be represented by *any one of the shortest such plans* or in more formal terms, pSTRIPS planning is amendable to *dynamic programming* (DP)[Dre02]. DP is special *divide and conquer* technique applicable when a problem can be broken down into "slightly smaller" subproblems which overlap (see [DPV06]) and allows to significantly reduce the number of candidates that have to be considered during the search. Its *bottom-up* form consists of initially solving the smallest class of subproblems in a domain, memoizing their solutions and use these to incrementally solve (and memoize) the next harder class of subproblems until one ends up with a solution for the original problem.

2.3.2 Breadth-first search

To recapitulate, equivalence classes of any STRIPS instance can be represented by a pair of a domain-state and plan $\langle s, \pi\rangle$ where $s = \delta(i, \pi)$ and there is no shorter sequence π' with $s = \delta(i, \pi')$. Beginning from i, the idea is to first generate all plans for length one and store all discovered equivalence classes. One can then generate the equivalence classes with plan length $d + 1$, by iterating over all classes with length d $\langle s, \pi\rangle_d$, determining successors $s' = \delta(s, o)$ and, if s' is not already forming an equivalence class, add $\langle s', \pi \circ o\rangle_{d+1}$ to the set of classes. The

Algorithm 3: BFS-DD

Breadth-first search with duplicate detection;

Output: an optimal i-plan or $\perp$

$EC \leftarrow \{\langle i, [] \rangle_0\}$;
$d \leftarrow 0$;

while $(\forall s \forall \pi) \exists \langle s, \pi \rangle_d \in EC$ **do**
  foreach $(\forall s \forall \pi) \langle s, \pi \rangle_d \in EC$ **do**
    foreach $o \in O$ *s.th.* $\delta(s, o) \neq s$ **do**
      $s' \leftarrow \delta(s, o)$;
      if $s' \in G$ **then**
        return $\pi \circ o$;
      end
      if $(\forall \pi' \forall i) \nexists \langle s', \pi' \rangle_i \in EC$ **then**
        $EC \leftarrow EC \cup \{\langle s', \pi \circ o \rangle_{d+1}\}$
      end
    end
  end
  $d \leftarrow d + 1$;
end
return $\perp$;

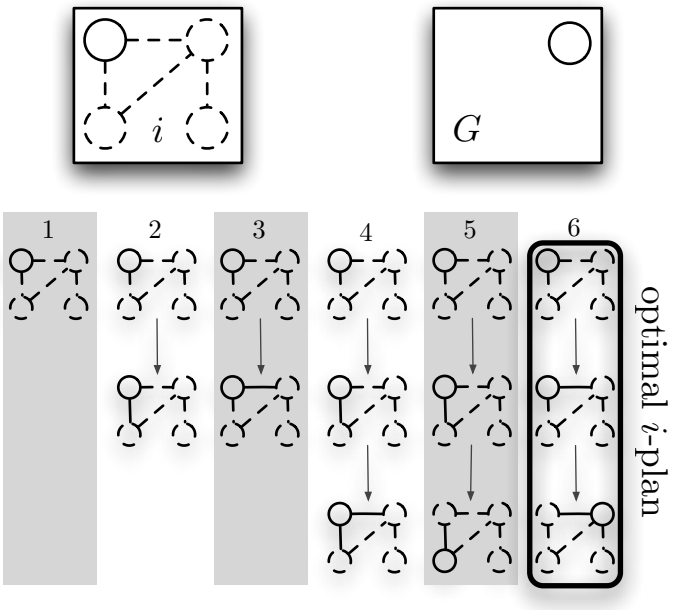


FIGURE 2.8 Candidates (worst-case) as generated by BFS-DD for an apartment instance where the agent starts in the kitchen with all doors closed and wants to move to the living-room.

process ends once a class of a goal state is created, with the corresponding π representing an optimal i-plan (π_i^*). Graph exploration methods in which states are visited in the order of their distance from the initial state are classified as *breadth-first* traversals. The specific algorithm described above is called *breadth-first search with duplicate detection* (BFS-DD) and a Listing is given in 3.

Figure 2.8 shows the candidates generated by BFS-DD for the example problem from Figure 2.7. Each candidate represents an equivalence class and hence results in a distinct endstate. Even for this trivial problem, the number of generated candidates is less than half that of IDDFS'. For larger problems, the difference is in general far more dramatic. Utilizing DP reduces the candidate set (or search space) from *all possible paths* down to the *shortest paths* to each state in the domain graph.

2.4 Terminology and Conventions

In the following, I want to introduce some terminology that is well established In the planning and scheduling community in connection with graph search algorithms.

IMPLICIT GRAPH A graph that is not explicitly represented in memory (e.g. through node

records cross-referenced by pointers), but through some *initial state* and a *generator function* that maps states to its successors.

CANDIDATE (OR STATE) GENERATION The process when a candidate or state is computed through application of the generating function.

CANDIDATE (OR STATE) EXPANSION The process of determining all applicable operators of the candidate followed by the computation of all corresponding "successors" by repeated application of the generating function on the candidate with each such operator.

OPEN At any state of the traversal (i.e. search), $Open$ denotes the collection of candidates or states that have been generated but not been expanded. $Open$ is often structured into classes of priority for state expansion according to the traversal's rules. In the context of this work, I indicate these by a subscript integers.

CLOSED At any state of the traversal (i.e. search), $Closed$ denotes the collection of candidates or states that have both been generated and expanded.

FRONTIER The class of $Open$ that by the traversal's rules has the currently *highest, equivalent* priority for expansion. In other words the candidates to be expanded next.

Listing 4 gives an example of the use of this terminology for BFS-DD. In it (and future listings), ancillary information for candidates or states is assumed to be stored in some associative data structure $Dict$. I will cover possible implementations of such structures in detail in chapter 4.

2.5 Heuristics

In search a *heuristic* h is some function that estimates the cost of optimal s-plans ($h(s) \approx c(\pi_s^*)$) for any state in the search space. The idea is to use such functions to rank states in $Open$ for expansion. The expansion order has a significant influence on the number of states processed during a search. For explicit state search, in the best case, only non-goal states that are part of some optimal i-plan are expanded (i.e. $|\pi_i^*| - 1$ states), in the worst case all non goal states have to be processed during construction of π_i^* (i.e. $|S| - |G| + 1$ states). Even in basic domains these bounds usually differ by orders of magnitudes. As an example, the (relatively) small 10-puzzle (5x2) domain [SS06] features a search space 2^P of around 1.8 million legal states, while the optimal plans for the two hardest instances comprise of 55 moves (actions)

Algorithm 4: BFS-DD with search terminology

Breadth-first search with duplicate detection;

Output: an optimal i-plan or $\perp$

$Open_0 \leftarrow \{i\}$;
$Closed \leftarrow \emptyset$;
$Dict[i] \leftarrow []$;
$d \leftarrow 0$;

while $Open \neq \emptyset$ **do**
  foreach $s \in Open_d (\equiv Frontier)$ **do**
    $Open_d \leftarrow Open_d \setminus \{s\}$;
    foreach $o \in O$ *s.th.* $\delta(s, o) \neq s$ **do**
      $s' \leftarrow \delta(s, o)$;
      if $s' \in G$ **then**
        return $Dict[s] \circ o$;
      end
      if $s' \notin Open \cup Closed$ **then**
        $Dict[s'] \leftarrow Dict[s] \circ o$;
        $Open_{d+1} \leftarrow Open_{d+1} \cup \{s'\}$;
      end
    end
    $Closed \leftarrow Closed \cup \{s\}$;
  end
  $d \leftarrow d + 1$;
end
return $\perp$;

each. In the heuristic search literature, this state evaluation is usually given in the following form.

$$f(s) = g(s) + h(s) \tag{2.1}$$

The evaluation for a state s, $f(s)$ is defined as the sum of the cost for reaching s from the initial state i, $g(s)$ and the cost of reaching the closest goal state from s as predicted by the heuristic estimator $h(s)$. Note that g and hence f are not functions as the evaluation is usually not fixed during a search. For the principled application of heuristics, two formal properties, *admissibility* and *consistency* are of special importance.

2.5.1 Admissibility

Estimators that represent *optimistic* guesses or in other words, whose values lower-bound (i.e. *never* overestimate) the cost of the (an) optimal s-plan for every state in a domain are called *admissible*.

$$h(s) \leq c(\pi_s^*) \quad \forall s \in S \tag{2.2}$$

As the denomination already alludes, *admissibility* (of the heuristic) is a central property for many optimal *informed* search algorithms. A search *algorithm* is considered *admissible* if it guarantees to find a minimal path (i.e. optimal plan) to a goal state, whenever such a solution exists.

2.5.2 Best-first search and A^*

The state evaluation is used to structure $Open$ into subsets. Candidates are then expanded in *ascending order of their evaluations*. The difference from BFS-DD is that states are not expanded in monotonic order of their distance from i. As noted, this evaluation is not a function, i.e. the evaluation can change during the traversal if a shorter paths to nodes previously expanded states are encountered. This complicates the algorithm slightly, as such states have to be *reopened* (i.e. removed from $Closed$ and reentered into $Open$ according to their updated, more promising evaluation). The resulting A^*-algorithm is given in Listing 5.

Figure 2.9 shows the candidates generated by A^* for the example problem from Figure 2.7. The simple heuristic in the example estimates the necessary number of actions by only considering the living room doors and whether the agent is in the living room. It is given as

Algorithm 5: A^*

A^* - best-first search with duplicate detection;

Output: an optimal i-plan or $\perp$

$Open_{h(i)} \leftarrow \{i\}$;
$Closed \leftarrow \emptyset$;
$Dict_\pi[i] \leftarrow []$;
$Dict_{depth}[i] \leftarrow 0$;
$d \leftarrow 0$;
while $Open \neq \emptyset$ **do**
  foreach $s \in Open_d (\equiv Frontier)$ **do**
    $Open_d \leftarrow Open_d \setminus \{s\}$;
    if $s \in G$ **then**
      return $Dict_\pi[s]$;
    end
    foreach $o \in O$ *s.th.* $\delta(s, o) \neq s$ **do**
      $s' \leftarrow \delta(s, o)$;
      if $s' \in Open \wedge Dict_{depth}[s'] > d + 1$ **then**
        $Open \leftarrow Open \setminus \{s'\}$;
      end
      if $s' \in Closed \wedge Dict_{depth}[s'] > d + 1$ **then**
        $Closed \leftarrow Closed \setminus \{s'\}$;
      end
      if $s' \notin Open \cup Closed$ **then**
        $Dict_\pi[s'] \leftarrow Dict_\pi[s] \circ o$;
        $Dict_{depth}[s'] \leftarrow d + 1$;
        $Open_{d+1+h(s')} \leftarrow Open_{d+1+h(s')} \cup \{s'\}$;
      end
    end
    $Closed \leftarrow Closed \cup \{s\}$;
  end
  $d \leftarrow d + 1$;
end
return $\perp$;

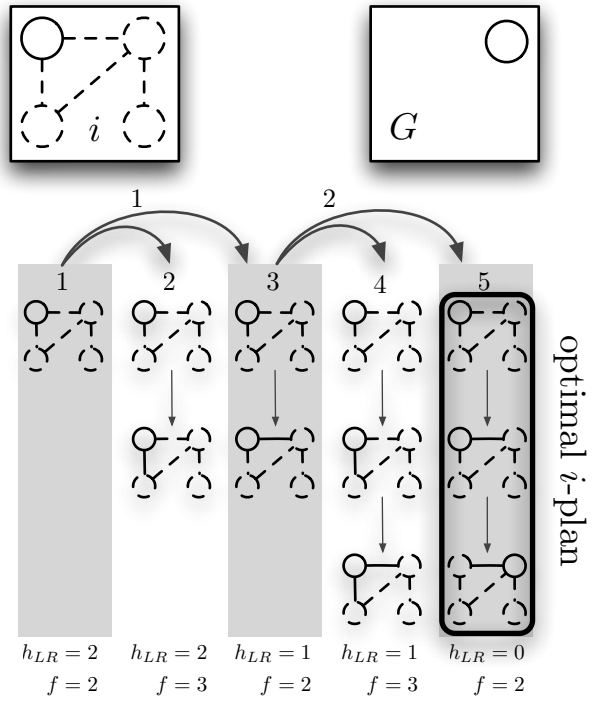


FIGURE 2.9 Candidates (worst-case) as generated by A^* for an apartment instance where the agent starts in the kitchen with all doors closed and wants to move to the living-room. Heuristic h_{LR} comprises of three possible values 0 if the agent is in the living room, 1 if the agent is not in the living room but any living room door is open and 2 otherwise.

$$h_{LR}(s) = \begin{cases} 0 & \text{if } p_C \in s \\ 1 & \text{if } p_C \notin s \wedge \{p_1, p_2, p_3\} \cap s \neq \emptyset \\ 2 & \text{else} \end{cases}$$

Heuristic h_{LR} codifies two pieces of domain knowledge, namely that changing rooms requires at least one move action and accessing a room requires at least one of its doors to be open. This construction makes h_{LR} trivially admissible.

In comparison to BFS-DD, A^* with h_{LR} needs only two expansions, which on this small example does not vastly influence the number of generated candidates. For larger problems however, a good heuristic usually reduces the number of state generations by multiple orders of magnitudes over the unguided case.

A^* is *admissible*, if used with an admissible heuristic. The proof is straightforward. By the time A^* terminates with some π_i^*, its construction guarantees that all candidates with a *lower* evaluation than π_i^* have been generated and tested. As the evaluation *lower-bounds* the "true cost" of any candidate π, any untested candidate π' with $f(\pi') > |\pi_i^*| (\equiv c(\pi_i^*))$ cannot be an optimal i-plan. On a side-note, if one uses $h(s) = 0, \forall s \in S$ (clearly an admissible heuristic), A^* "nearly" degenerates into BFS-DD[3], which is by this argument also admissible.

2.5.3 Consistency

A stronger attribute for heuristics is *consistency*. Formally, a heuristic function is consistent if two properties hold for all states of the domain graph.

$$h(s) \leq c_{s \to s'} + h(s') \quad \forall s \in 2^P, \forall s' \in \delta(s) \tag{2.3}$$

$$h(g) = 0 \quad \forall g \in G \tag{2.4}$$

That is, first, the estimate of any successor of a state must not exceed the estimate of a state plus the cost of the action to generate the successor and, second, the estimate for all goal states must be 0. The first property demands that for all state-successor pairs, $h(s) - h(s') \leq c_{s \to s'}$ must hold and it thus *never* overestimates the cost between any two states in the search space, which can be understood as a form of *triangle inequality*. For this reason, the property (and such heuristics) is also referred to as *local admissibility*. Consistency is traditionally defined

[3] the difference is in when a state is tested against the goal condition: immediately after it is generated for BFS-DD or before it is expanded for A^*

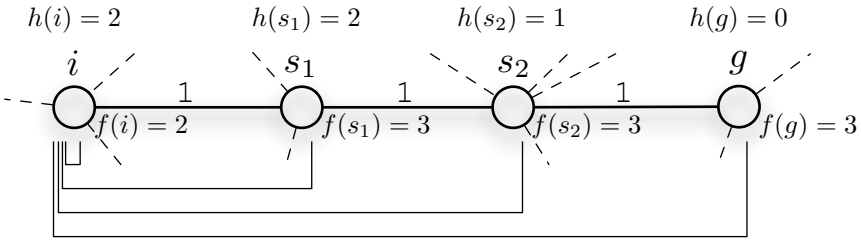


FIGURE 2.10 Example plan showing the monotonicity of f for a *consistent* heuristic h.

over all pairs of states (as opposed to the state-successor relationship above) in the domain graph along with the cost of transitioning between these states. The corresponding interpretation of that definition is that the *triangular inequality* holds for heuristic estimates of elements of the domain graph. Pearl introduced the above definition and showed its equivalence [Pea84]. I follow his definition as it is often easier to verify in practice and allows (in my opinion) for a more intuitive interpretation of the nature of consistency (as also noted by [Hol10]).

It implies that cost estimates (through f) of partial solutions are *monotonically* increasing in distance from i (see figure 2.10 for an example) or formally

Theorem 1. *Consistent heuristics make f monotonically increasing in the search depth.*
$f(s') \geq f(s) \quad \forall s \in S, \forall s' \in \delta(s)$

Proof. [Nil98]

$$\begin{aligned} f(s) &= g(s) + h(s) \\ &= g(s') - c_{s \to s'} + h(s) \\ &\leq g(s') + h(s') \\ &= f(s') \end{aligned}$$

□

Another characteristic of *consistent* heuristics is that they are also *admissible*. This follows straightforwardly from the second property and the local admissibility. The reverse does not hold in general. Consistency is important property in combination with A^* - with a consistent heuristic, the monotonic order of candidate evaluation and expansion guarantees that no candidate will be reopened. In fact, it can be shown that with a consistent heuristic, A^* is *optimal* in regards to the number of candidate evaluations and expansions for a given problem and heuristic among best-first algorithms [DP85]. In other words, there is *no* admissible best-first

algorithm that using the *same heuristic information*, can do with expanding less candidates than A^*[4].

A popular technique for an admissible, yet inconsistent heuristic to exchange heuristic information between states and successors is the *pathmax* technique introduced in [Mer84]. Here estimates h are updated to h' through repeated applications of the *pathmax* equation 2.5.

$$h'(i) = h(i) \tag{2.5}$$

$$h'(s') = \max(h(s'), h'(s) - c_{s \to s'}) \quad \forall s \in S, \forall s' \in \delta(s) \tag{2.6}$$

This (by its definition) achieves monotonically increasing cost estimates along paths represented in $Open$ and $Close$ for an *admissible* heuristic h . The technique works by propagating h's estimates along the search graph, taking into account the problem's action costs. Technically h' is *consistent*[5]. The second property follows directly from the admissibility of h, whereas *local admissibility* is a result of the definition of *pathmax*.

Theorem 1. *Inductive updates through pathmax transform an admissible heuristic h into a locally admissible heuristic h' (cf.[RNC+10])*
$h'(s) \leq c_{s \to s'} + h'(s') \quad \forall s \in S, \forall s' \in \delta(s)$

Proof.

$$\overbrace{h'(s') \leq h'(s) - c_{s \to s'}}^{\text{If } h'(s') = h(s')} \quad \vee \quad \overbrace{h'(s') = h'(s) - c_{s \to s'}}^{\text{Else}} \tag{2.7}$$

□

However while this information dissemination can lead to substantial reductions in state expansions (see [Mer84] and [ZSH+09]) it does *not* guarantee (as a *genuinely* consistent heuristic would) that A^* will never have to reopen states (see [ZH02] and [Hol10]).

[4] [Hol10] notes that this argument technically ignores the case where multiple states with $f(s) = c(\pi^*)$ are encountered and also that expanding less nodes does not necessarily imply less runtime.
[5] Note that this also implies the *admissibility* of h'.

CHAPTER 3

Target Value Search

A domain that nicely showcases the merits (and limits) of different search algorithms and heuristics as well as the complexities arising from removing some of the restrictive assumptions of classical planning are *target-value-path* problems. A target value-path problem is to find a path between two nodes in a graph, such that its valuation $g(\pi)$ (typically the sum of the path's edge values) comes as close as possible to the target value. Nykänen and Ukkonen [NU99], [NU02] concerned themselves with a similar decision problem, namely whether paths of a given cost exist between any two nodes of the graph.

3.1 Example Domains

3.1.1 Pervasive Diagnosis for manufacturing systems

Target value problems arise for example when integrating model-based planning and diagnosis as in pervasive diagnosis (see [Fro02], [KPdK+08] and [KPD+10]). This work originated within PARC's *Tightly Integrated Production Printer* (TIPP) project [WDZF11], a planner-controlled, hypermodular manufacturing system comprising of reconfigurable and self-describing modules such as paper routers, inverters, print engines, feeders and finishers.

In this context, a diagnosis engine represents its belief in the malfunctioning of each component with a probability value (i.e. 1.0 denotes a component that is known to be malfunctioning and 0.0 denotes one that is known to be fine). After each plan execution, the engine updates its beliefs based on the fitness of the resulting product and the involved components. The idea is to exploit degrees of freedom in production plans for such manufacturing systems (see figure 3.1) to gain information about the health of its components and use this information to increase long-run productivity.

From a search perspective, this problem presents itself as a graph modeling the structure

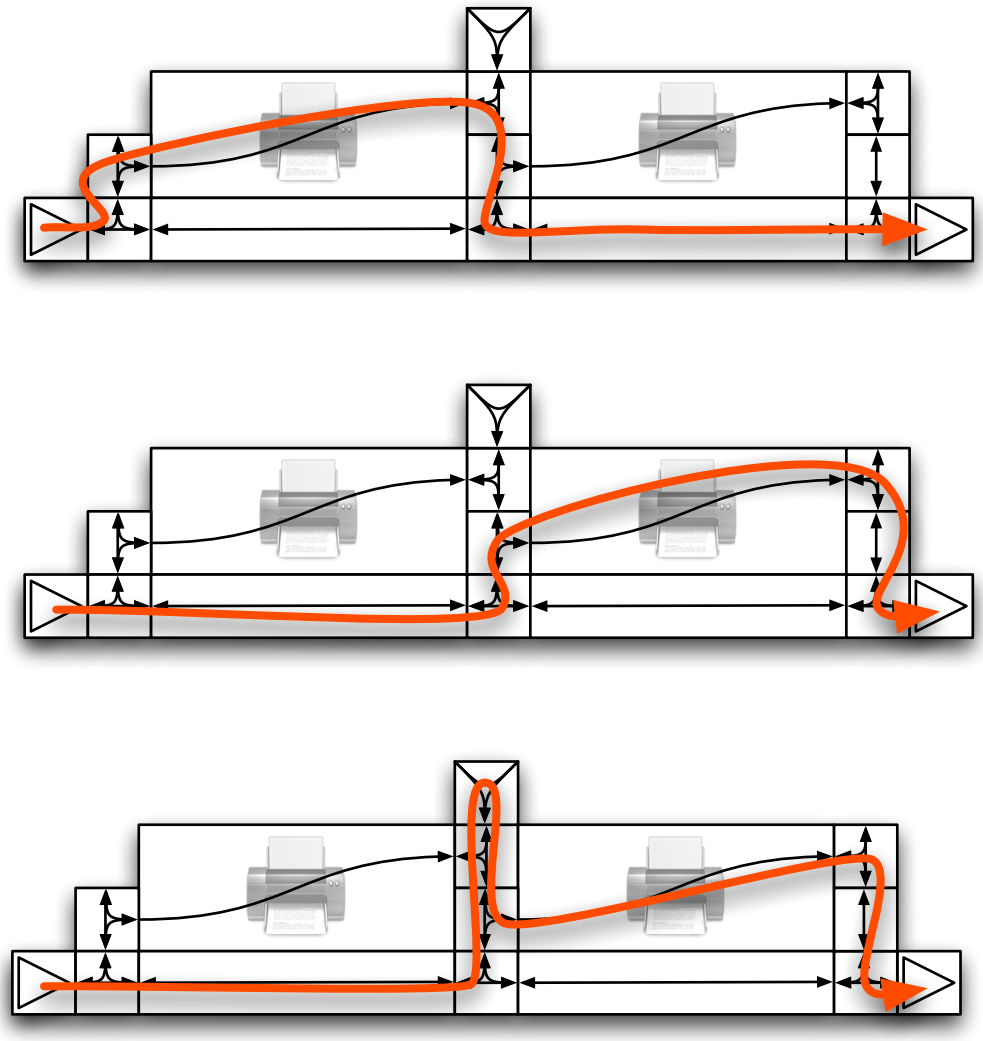


FIGURE 3.1 Conceptual representation of a modular printing system and (some) possible ways of printing a single-sided sheet.

of the production process with states corresponding to intermediate products, edges to operations of individual system components and edge weights representing the diagnosis engine's confidence in the operability of the respective component. For the following, I will constrain the problem to its simplest case, that of single, non-intermittent faults. Here one assumes at most one component in the production system is malfunctioning and if such a component is involved in a production plan, the resulting product will be faulty. Furthermore, such manufacturing defects can only be discovered in the final product. Hence, the problem is slightly beyond what can be expressed in classical planning frameworks (c.f. chapter 2.1.3) as the system is neither *fully observable* (assumption **A3**) nor does it sport straightforward *attainment goals* (assumption **A5**).

Executing plans whose predicted success probability (based on the system's current beliefs) is as close as possible to 0.5 will maximize the diagnostic engine's information gain about the system's true state [LdKK+08]. Intuitively this is because such plans involve components the diagnosis engine knows least about. Due to the assumption of non-intermittent faults, the probability (based on the current beliefs of the diagnosis engine) for manufacturing a faulty product is (perhaps un-intuitively) the sum of the malfunctioning probabilities of all components involved in the plan. Note that this sum never exceeds 1 due to the single fault assumption (for an in-depth discussion of the theoretical background and cases where these restrictions are lifted, see [LdKK+08] and [KPD+10]). For brevity, I here assume that the process graph is suitably preprocessed to remove (or unroll) cycles.

3.1.2 Consumer Recommender Systems

Another example of a target-value-path problem is a recommender system for recreational hikes in a National Park (see figure 3.2). In this domain, a hiker specifies his parking spot and the desired hike duration (as well as possibly some landmarks he would like to see) and the system recommends an appropriate hike based on these parameters. The system uses these inputs and a map of the area with expected traversal times for each path segment to first generate an unrolled graph incorporating all cycle-free paths beginning and ending at the parking spot and visiting the specified landmarks and then runs a target-value search on the this graph to find the hike that best matches the parameters.

Other potential domains include comprehensive training programs, with complex temporal and causal interdependencies between courses where participants need to reach certain point thresholds (i.e. university studies or mandatory professional training programs). Another interesting area are automated quality assessment systems for software development. In this domain, one of the problems is to determine nightly-build processes as selections out of a

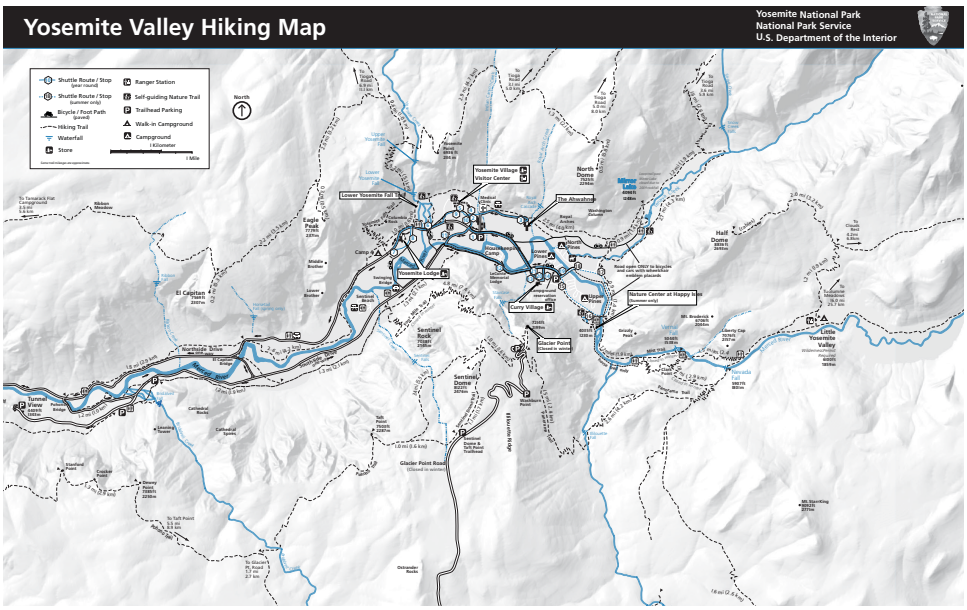
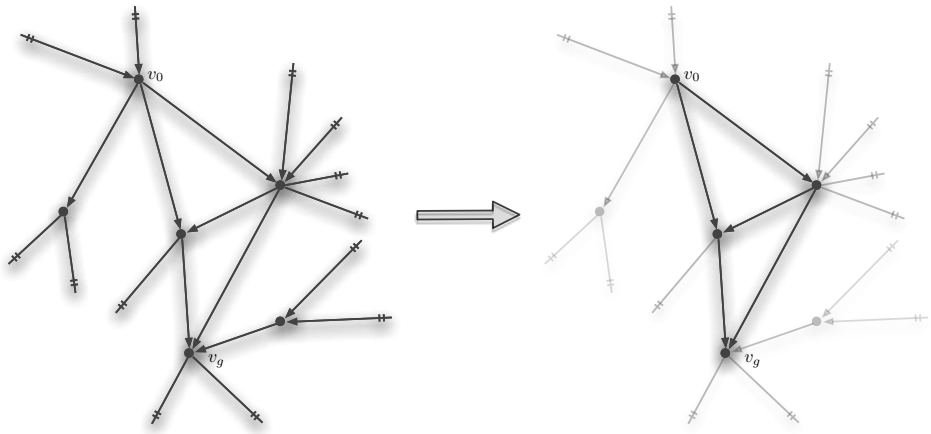


FIGURE 3.2 Hiking map of the Yosemite Valley National Park.

large set of interdependent transformation (compilation, automated refactorings, code generation, etc.) and analysis tasks (unit and integration tests, code coverage, model checking, clone detection, profiling, etc.) such that they make best use of available time between development activities.

3.2 Problem definition

Abstractly the target value search problem is defined as follows. Given a directed acyclic graph $G = (V, E)$ with non-negative edge values where the set of vertices V corresponds to the problem states, each edge $e \in E$ between vertices $v, v' \in V$ corresponds to a possible transition between problem state v and v' through some domain operator and the edge-value to the value associated with that transition ($g(v \to v')$), a *target-value-path* between two vertices $v_0, v_g \in V$ with *target-value* tv (or plan $\pi^{tv}_{v_0 \to v_g}$ in short, respectively π^{tv} when the initial and end vertex are clear in the context) is a sequence of edges leading from v_0 to v_g, whose value is as close as possible to tv. The value $g(\pi)$ of a plan π is simply defined as sum of its comprising edges' values. Let $\prod_{v_0 \to v_g}$ be the set of all plans leading from v_0 to v_g in G. Then $\prod^{tv}_{v_0 \to v_g} = \underset{\prod_{v_0 \to v_g}}{\operatorname{argmin}}(|tv - g(\pi)|)$, the set of paths between v_0 and v_g with minimal deviation from the target-value is defined as the *target-value path set* with respect to v_0, v_g, tv and every element of $\prod^{tv}_{v_0 \to v_g}$ is a target value path. In the following *target-value search* (or tvs, in short)



Original graph G Connection graph C

FIGURE 3.3 Excerpt of the search graph G and the corresponding connection graph C for vertices v_0 and v_g.

refers to function mapping tuples $\langle v_0, v_g, tv \rangle$ (in a non-deterministic way) to some element of $\prod^{tv}_{v_0 \to v_g}$.

3.2.1 Conventions

For reasons of clarity and brevity, I limit the following discussion to *connection graphs*. The connection graph C_{v_0,v_g} is the subgraph of G containing v_0, v_g and those vertices in V that are both descendants of v_0 and ancestors of v_g as well as all edges ($\in E$) between them. In other words, I assume that all vertices that either cannot be reached from v_0 or from which v_g cannot be reached are pruned from G, along with corresponding edges. Note, that C can be extracted by creating the respective intersection of vertex and edge sets reachable through breadth-first sweep from v_0 along successor links and from v_g along predecessor links, in time and space linear in the size of G (hence $O(|V|+|E|)$). For an example, see figure 3.3. I generally assume that edge values are positive and, as the graphs are explicit, that predecessors can be accessed efficiently.

Here, a few remarks on terminology and notation in this chapter. I omit indices where they are implied by context. I use the term *prefix* for any path beginning at v_0 (to some vertex in C), the term *suffix* for any path (from some vertex in C or interchangeably from (the last vertex of) some prefix) ending in v_g; note that due to the construction of C, any vertex or prefix will have at least one suffix. *Valuations* are defined for all paths in C as the sum of their respective edge values. *Costs* are defined only for paths in $\prod_{v_0 \to v_g}$ as the *absolute difference between valuation and target-value*. The term *completion* of a prefix denotes its concatenation

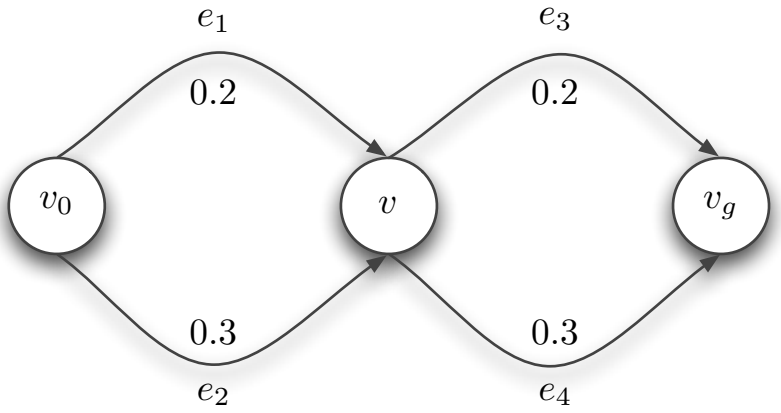


FIGURE 3.4 Target value search does *not* exhibit *optimal substructure*. Consider the above graph for $tv = 0.5$. After expanding v_0, one has two paths $[e_1], [e_2]$ to v_1. Both can lead to optimal solutions with the right completion (i.e. $[e_1, e_4]$ and $[e_2, e_3]$), the selection of which *depends on the whole prefix, not only on its last vertex.*

with any of its suffixes, the term *optimal completion* of a prefix w.r.t a target-value, denotes a completion that brings the valuation of the combined path (exactly) to the target-value (i.e. cost zero) and the term *best completion* for the element that brings it *closest* to the target-value (i.e. minimal cost). Paths (or variables holding paths) are denoted with symbol π - the use subscripts with arrow notation $\pi_{a \to b}$ implies that following the path (or executing the plan) from vertex (in state) a leads to (ends up in) b.

3.2.2 Complexity

At first glimpse, the problem might appear unchallenging. In non-cyclical connection graphs, satisficing planning is trivial. For example, a depth-first search (c.f. section 2.3.1) in such a structure is complete even without backtracking. Also the graphs I concern myself with are generally rather small (i.e. such that there is no problem in storing the entire graph in memory). The combinatorial search problem is the (seemingly simple) selection of the best solution from a set of solutions represented by the connection graph.

The challenge stems from the fact that target value search does not exhibit the property of *optimal substructure*, a prerequisite for *greedy* or *dynamic programming* approaches (as are exploited in shortest-path problems, c.f. section 2.3.1.1). Consider an arbitrary decomposition of a target-value path

$$\pi^{tv}_{v_o \to v_g} = \overbrace{\pi^{tv'}_{v_o \to v}}^{\text{Prefix}} \circ \overbrace{\pi^{tv''}_{v \to v_g}}^{\text{Suffix}}$$

The partitions' target-values tv' and tv'' are interdependent as $tv' = tv - g(\pi^{tv''}_{v \to v_g})$ and $tv'' = tv - g(\pi^{tv'}_{v_o \to v})$ and so are the respective *cost functions* for the subproblems. See figure 3.4 for an example. Also the number of solutions represented by C is in general exponential in the size of the graph. As in the worst case, all prefixes in C up to (roughly) the target-value will have to be generated during a target-value search, the problem is in $EXPTIME$. In their treatise [NU99] of the related decision problem, Nykänen and Ukkonen prove that the corresponding decision problems are generally NP-complete and specifically show that for the simplest case of integer edge-weights, pseudo-polynomial algorithm exist (i.e. that those problems are weakly NP-complete). NP-equivalence of target-value search problems then follows directly from the equivalent decision problem in cycle-free graphs being in NP.

3.3 Heuristics for Target Value Search

3.3.1 A straightforward approach

A intuitive approach to tackle target-value search problems with heuristic search is to use some estimate h of suffix lengths and search through *path space* with A^* using an *inadmissible* evaluator, such as $f(\pi_{v_0 \to v}) = |g(\pi_{v_0 \to v}) + h(v) - tv|$. Note that due to the lack of optimal substructure, the elements of our search are *not* (as usual) the vertices of the graph, but the prefixes (i.e. entire edge sequences relative to v_0). Also using an admissible heuristic h does not result in an admissible evaluator f due to the *non-linearity* introduced by the absolute value operator (see figure 3.5 for an example). Because of this, the A^* will be complete but inadmissible and solutions will usually not be generated in *descending order of their quality*.

The idea then is to adapt the algorithm to continue with the search until one can verify the optimality of a produced solution. This can be done as follows. Once the A^* produces the first solution $\pi^{tv}_{\approx}$ it is set as current best solution and the search continues with some small modifications. First, based on the current best solution, $Open$ is pruned of all prefixes whose g value exceeds $tv + |tv - g(\pi^{tv}_{\approx})|$ (i.e. all prefixes that can due to their already accrued sum of edge values only result in worse solutions). Second, all generated prefixes are tested (and potentially pruned) against the current best solution. Third, if a new solution is produced that is closer to the target-value, it replaces the current best solution. The search terminates if either a perfect solution is found (i.e. $g(\pi^{tv}_{\approx}) = tv$) or $Open$ is empty (returning $\pi^{tv}_{\approx}$).

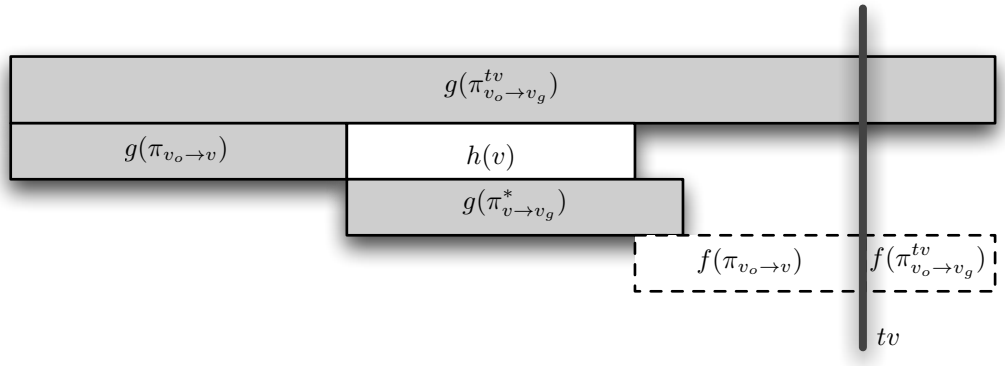


FIGURE 3.5 An example showing why an admissible heuristic for classical planning is not admissible for target value problems. A suitable estimator of the cost (i.e. deviation of the target value) of a prefix is $f(\pi_{v_o \to v}) = |tv - g(\pi_{v_o \to v}) + h(v)|$. The upper bar represents the value of the target value path $\pi^{tv}_{v_o \to v_g}$. Below is some prefix $\pi_{v_o \to v}$ (gray bar) along with an admissible estimate $h(v)$ for the distance between v and v_g (white bar). Below is the actual shortest distance between v and v_g in the graph. The last bars give the evaluators for $\pi_{v_o \to v}$ and $\pi^{tv}_{v_o \to v_g}$. Notice how due to the nonlinearity, $f(\pi_{v_o \to v})$ overestimates the optimal path cost $f(\pi^{tv}_{v_o \to v})$.

3.3.2 An Admissible Estimator for Target Value Search

Compared to an exhaustive search, the above scheme can help to reduce search times significantly (c.f.experimental results later in this chapter). Much preferable however would be to design an admissible estimator for these problems, immediately directing the search towards an optimal solution. In somewhat related work, Dow and Korf show how an admissible heuristic for best first-search can be constructed for the *treewidth* problem, which also sports a *non-additive* cost function [DK07].

In the following, I will describe such an heuristic that we developed at PARC. The idea behind it is to concisely annotate each vertex v in the graph with information about the values of all elements in $\prod_{v \to v_g}$ (in other words, the set of suffixes from v in C). In the initial version, each annotation takes the form of an interval, representing the range of values of these suffixes (see [KPdK+08] and [KSP+08]). Now this interval $[l_v; u_v]$ can be leveraged for any given prefix $\pi_{v_0 \to v}$ and target value tv by first computing the optimal target value for the completion $tv_{suf} = tv - g(\pi_{v_0 \to v})$ and then derive an admissible estimate for the cost of the best completion

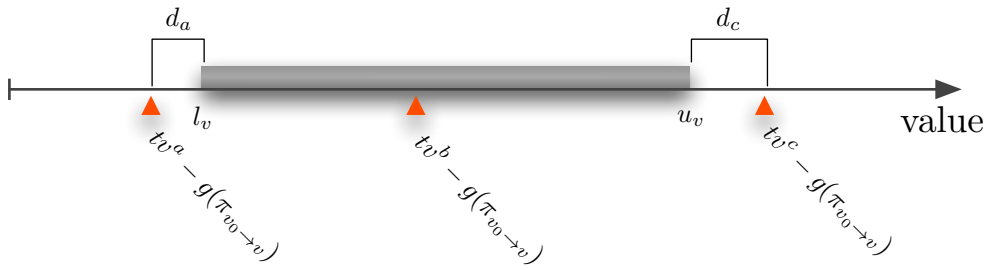


FIGURE 3.6 Heuristic computation for a prefix $\pi_{v_0 \to v}$ and three different target values tv^a, tv^b and tv^c corresponding to the three cases in equation 3.2. The gray bar $[l_v; u_v]$ represents the interval of completion valuations for vertex v. For tv^a, the optimal completion target value falls below that interval, i.e. the best completion will at least have cost d_a. For tv^b, the optimal completion target value falls inside the interval, i.e. there *might* be an optimal completion in C. For tv^c, the optimal completion target value falls above the interval, i.e. the best completion will at least have cost d_c.

$$f(\pi_{v_0 \to v}) = \begin{cases} l_v - tv_{suf} & \text{if } tv_{suf} \leq l_v \\ 0 & \text{if } l_v < tv_{suf} < u_v \\ tv_{suf} - u_v & \text{else} \end{cases} \tag{3.1}$$

The second case suggests, that an optimal completion might exist in the graph, hence the heuristic value is set to 0. In the first and last case, one can derive that no optimal completion exists and that the best completion is the suffix with the *highest* respectively *lowest* valuation. Figure 3.6 depicts these relationships graphically.

An important observation is that if the bounds l_v and u_v represent the actual valuations of the lowest and highest valued path in $\prod_{v \to v_g}$, f gives the *actual costs* of the prefix and its best completion in cases one and three (i.e. a perfect estimate). From now on, I will assume that this is the case. The second case represents an optimistic guess as the interval usually represents the valuations of a very large number of completions. To guarantee admissibly, one has to account for the possibility that an optimal completion can be found amongst these suffixes. Figure 3.7 gives an example for the comparative evaluation of two prefixes in the frontier of a best-first search, where to guarantee admissibility on has to prefer the (in hindsight) "wrong" candidate.

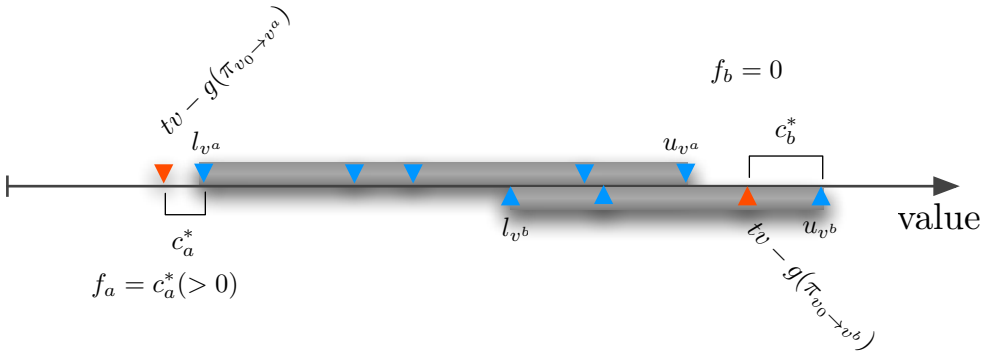


FIGURE 3.7 Valuation of two candidate prefixes $\pi_a = \pi_{v_0 \to v^a}$ and $\pi_b = \pi_{v_0 \to v^b}$ in the frontier of a best-first search. The upper triangles represent the valuations of all completions of v^a, the lower ones of v^b. If this (complete) information was available, π_a would be preferred to π_b as its best completion results in lower cost (i.e. $c_a^* < c_b^*$). As $tv - g(\pi_b)$ falls within v^b's interval the (admissible) estimate f_a exceeds f_b and hence the heuristic prefers π_b.

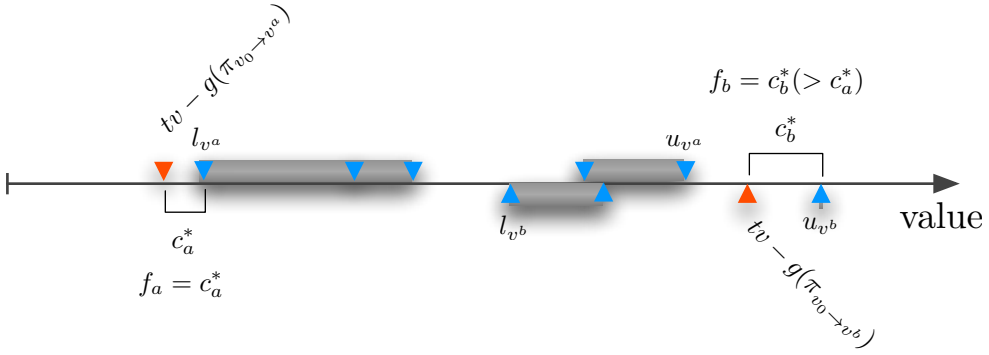


FIGURE 3.8 The situation from figure 3.7 with a 2-interval heuristic. Now f_b exceeds f_a and the heuristic (correctly) prefers π_a.

3.3.3 Multi-interval Heuristic for Target Value Search

To reduce this problem, I expand the annotations of C's vertices to *multiple, disjoint* intervals. To limit the growth of this precomputed associative store, I bound the maximum number of intervals per vertex by a constant k. Varying k allows to tradeoff space and heuristic quality. With a suitable construction algorithm for computing the intervals, heuristics with higher k dominate those with lower values (i.e. $\forall \pi \in 2^E \wedge k_a > k_b, f^{k_a}(\pi) \geq f^{k_b}(\pi)$). Figure 3.8 gives an example of how a higher k estimator can improve guidance during a best-first search (compared to figure 3.7). Intuitively, the smaller the aggregate ranges covered by the intervals of some vertex v's interval set, the higher the chance for a corresponding prefix $\pi \in \prod_{v_0 \to v}$ to be estimated as $f(\pi) = f^*(\pi)$ (i.e. as the cost of the prefix' best completion). Formally, given a

vertex v with *disjoint* intervals $[l_v^i; u_v^i]$, a given prefix $\pi_{v_0 \to v}$, target value tv and corresponding optimal completion valuation $tv_{suf} = tv - g(\pi_{v_0 \to v})$ the multi-interval heuristic is defined as

$$h(\pi_{v_0 \to v}) = \begin{cases} 0 & \text{if } \exists i, l_v^i < tv_{suf} < u_v^i \\ \min(\{|tv_{suf} - l_v^i|\} \cup \{|tv_{suf} - u_v^i|\}) & \text{else} \end{cases} \tag{3.2}$$

or intuitively as 0, iff tv_{suf} falls inside one of the intervals or the minimum distance between tv_{suf} and *any* interval bound otherwise. I will from here on refer to the collection of intervals of a vertex as its *entry* and the collection of entries as the interval *database* or *store*.

3.3.4 Computing the Interval Store

This database can be computed efficiently on the C through dynamic programming. I first turn to computing the entry of a vertex v given the entries of its successors $v'_1 \dots v'_n \in \delta(v)$. In a first step, the intervals of each v'_i are transformed by adding the value of the corresponding edge $g(v \to v'_i)$ to *all* bounds and added to v's entry. In a second step, one repeatedly looks for *overlapping* intervals in v's entry and combines them until all intervals are disjunct. Finally, as long as the number of intervals in v's entry exceeds the allowance k, one takes the two closest intervals and combines them. See figure 3.10 for an example. In the special case that a vertex has no predecessors, its entry is simply set to $\{[0; 0]\}$.

As C is a directed and acyclic graph, there is a well defined topological order (see [Jar60]) for vertices of the graph. Processing vertices in their inverse topological order guarantees that all successors of a vertex have been processed before the vertex itself is processed. This can be achieved as follows. The process starts by initializing a counter for each vertex v in C to the number of v's successors in C and adding the goal nodes to a *first-in first-out* (fifo) queue. While this queue holds nodes the next node is removed, its entry computed according to procedure above (see figure 3.10) and the counters of all its predecessors decremented by one. If one of these counters hits 0, the corresponding predecessor (vertex) is in turn added to the queue. An example is given in figure 3.10.

3.3.4.1 Complexity of computing the interval store

This technique guarantees that each vertex in the connection graph is processed only once. This can be shown through an induction proof: if all of the descendants of v (i.e. the transitive closure of the successor relationship) have been accessed only once, then this is certainly true for its successors, so v's counter will be 0 and v will be added to the queue and thus be processed once (induction step). v_g starts on the queue and has no descendants in C, so v_g will

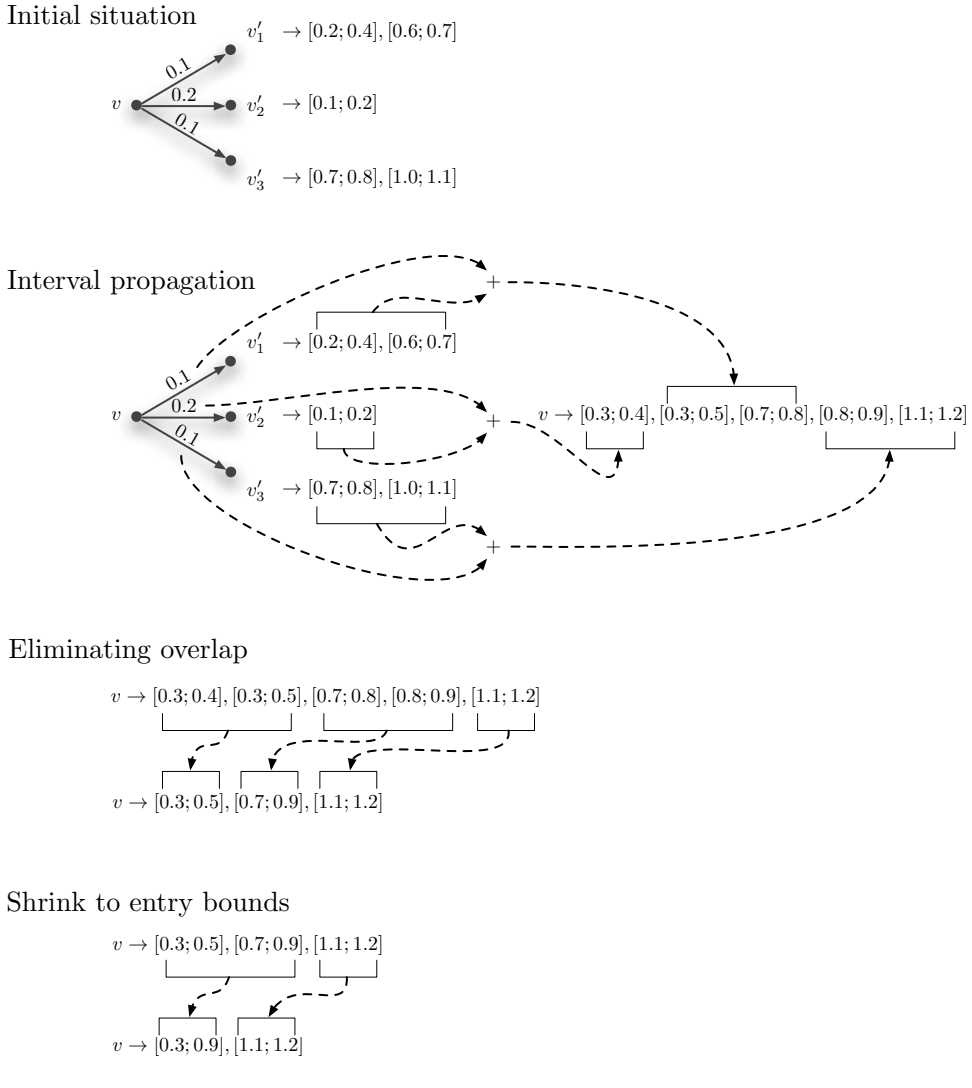


FIGURE 3.9 Construction of a database entry for vertex v given entries for its successors v'_1, v'_2 and v'_3.

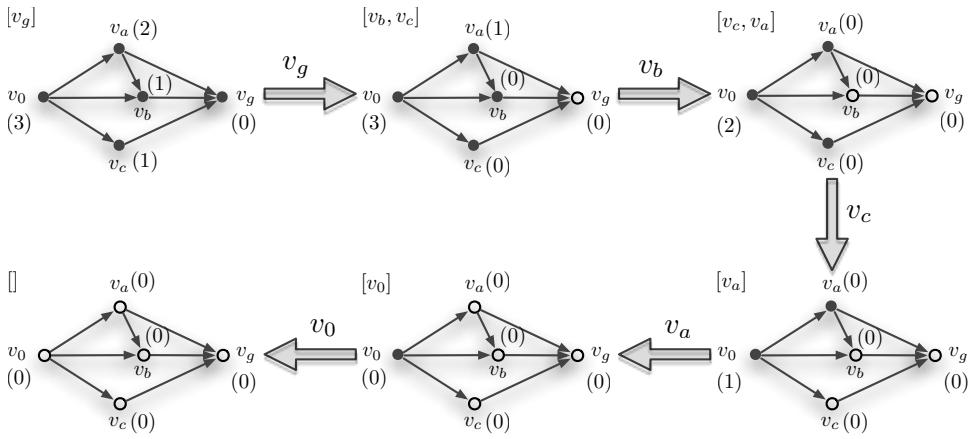


FIGURE 3.10 Construction of the database for a small connection graph of five vertices. Round brackets denote counters, square brackets the fifo queue. Filled vertices still need to be processed.

only be processed once (base case). From this it follows that each edge will be accessed twice (in the predecessor direction to decrement the predecessor count and, in the other direction to access successor intervals). This results in running time of $O((k+1)|E|)$, where k denotes the user defined maximum size of a entry. The worst-case space requirement of this algorithm is linear in the size of the vertex set of C. If the lattice graph shows little topological structure (i.e. all vertices other then v_0 and v_g have exactly v_0 as predecessor and v_g as successor) the queue can grow up to $|V|-2$ vertices in a lattice graph. Redeemingly, target-value search is trivial in such graphs, as the number of paths is only $|V|-2$.

3.4 Algorithms for Target Value Search

With an admissible estimator at hand, it is time to turn towards a suitable search strategy for target value search. Even though any such strategy must operate in path space (i.e. 2^V, c.f. section 3.2.2), candidate structure can still be leveraged in limited ways. First, prefixes ending in the same vertex *with equal valuation* form an equivalence class and can hence be considered duplicates. This is as the goal is to find any of the best solutions. Because any two equally valued prefixes to some vertex v will result in respective completions that are equal from a cost perspective, only one of them must be considered during the search. Second, the addition to guiding the search, the interval heuristic can to detect when the problem of finding completion for some prefix degenerates into a shortest or longest path problem. In a directed acyclic graph, these problems can be solved straightforwardly in vertex space.

3.4.1 Best-First Target Value Search

I want to start by summarizing the effects of the above properties of (multi-) interval estimators on the admissibility of search algorithms.

DUPLICITY the best completions of any set of prefixes ending in the *same vertex*, with equal prefix values are identical and hence the best target value paths for all of these prefixes will have equal deviation from the original target value. In other words, such sets form an equivalence class in which the respective best solutions stemming from all of its elements will be equal w.r.t. the target-value search's objective function. Because of this only one such prefix needs to be considered for admissibility.

DOMINATION For any prefix $\pi_{v_0 \to v}$ with $f(\pi_{v_0 \to v}) > 0$, $f(\pi_{v_0 \to v})$ *is* the actual deviation of the prefix's best completion from the target value (i.e. $f(\pi_{v_0 \to v}) = f^*(\pi_{v_0 \to v})$). From this one can deduce that for any pair of prefixes $\pi_{v_0 \to v_a}, \pi_{v_0 \to v_b}$ with $f(\pi_{v_0 \to v_a}) > f(\pi_{v_0 \to v_b}) > 0$, $\pi_{v_0 \to v_a}$ is *dominated* by (i.e. will necessarily produce a *worse* solution then) $\pi_{v_0 \to v_b}$ and can therefore be discarded without affecting admissibility.

Listing 6 shows how the interval heuristic can be integrated into an adaptation of the A^* algorithm called BFTVS. One important difference to the version presented in section 2.5.2 is that costs and their estimates (i.e. f) are *real-valued* as opposed to the discrete integer values for unit-cost operators in classical planning. Hence $Open$ here is a general priority queue that allows accessing its constituents in descending order of their priorities (usually fibonacci heaps [FT87] are the implementation of choice for these data structures as they efficiently support priority updates - a feature not required for BFTVS on which I will extend later). Another is that in this application (in contrast to shortest path search), it is entirely possible to have multiple paths between two nodes in $Open$ that are *not* mutually redundant because of *duplicity* and *domination*. Hence the entire path (predix) must be stored for elements in $Open$. For $Closed$ this is not necessary - it only serves the purpose of keeping track of which equivalence classes have already been processed in the search and these are defined by the vertex and path valuation alone. If a candidate is generated that corresponds to one of these classes, it can be safely discarded without affecting admissibility. After a candidate passes this test, $Open$ must be searched for an equivalent entry. The algorithm of 6 represents an equivlent, simplyfied version of what was given in [KSP+08] better showing the close relationship to A^*. Further simplifications are possible (and were used in the evaluation). Note that if any eqivalent entry exists $Open$, it will (by the above definition of equivalence) have equal priority. Because of this property, priority updates are unecessary and the implementation can be simplified by

Algorithm 6: BFTVS

Best-First Target Value Search

Input: C the connection graph
Input: tv the target value
Input: f an admissible estimator
Data: $cand$ a 3-tuple comprising of a vertex, prefix and target value $\langle v, \pi, tv \rangle$
Data: $Open$ a priority queue of $cand$ elements
Data: $Closed$ a set of vertex, target-value tuples
Output: a target value path or $\perp$

$Open$.push($\langle v_0, [], tv \rangle$,$f([])$);
while $cand \leftarrow$ *q.next()* **do**
  if $cand.v = v_g$ **then**
    return $cand.\pi$;
  end
  $Closed$.insert($\langle cand.v, cand.tv \rangle$);
  for $v' \in \delta(cand.v)$ **do**
    $tv' \leftarrow cand.tv - g(v \rightarrow v')$;
    $cand_{new} \leftarrow \langle v', cand.\pi \circ v \rightarrow v', tv' \rangle$;
    if $\neg$*Closed.has(*$\langle v', tv' \rangle$*)* **then**
      if $\neg$*Open.hasEq(*$cand_{new}$*)* **then**
        $Open$.insert($cand_{new}$, $f(cand_{new}.\pi)$);
      end
    end
  end
end
return $\perp$;

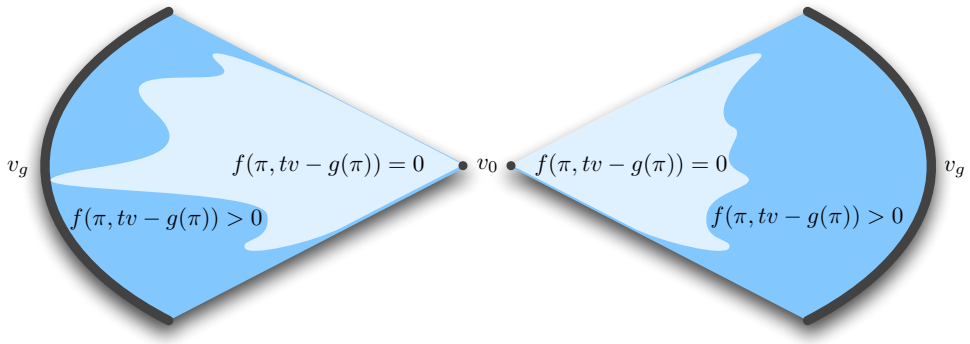


FIGURE 3.11 Division of the search space into guided (dark blue) and unguided (light blue) parts in case an optimal target value path exists (left) or not.

upon discovery of a new equivalence class, adding the respective class to *Closed* (which is then somewhat misleadinginly named), implement *Open* as a simple linked-list ordered by increasing f and restrict duplicate detection to *Closed*. In contrast to first approach (c.f. section 3.3.1), which in many cases (if there is no optimal target value path) has to generate all prefixes in C, valued less than $tv + f(\pi^{tv}_{v_0 \to v_g})$, BFTVS can often make do with a small subset which usually well compensates the cost for constructing the database (especially if it can be reused, i.e. multiple queries are performed with the *same* v_g). However in the worst case, both algorithms have to generate all prefixes with values $\leq tv + f(\pi^{tv}_{v_0 \to v_g})$ in C, a majority of which can end up in *Open* at same time. Worst-case memory requirements are hence exponential in problem size (i.e. C), which severly limits scalability. In practice, the issue can be somewhat attenuated for BFTVS with interval heuristics by exploiting the domination property. Admissibility is still guaranteed if from all candidates with estimated costs > 0 only the one with the lowest estimate is kept. Nevertheless, growth of *Open* and (to a lesser extend) *Close* is only exponentially bounded by the size of the connection graph.

3.4.2 Depth-First Target Value Search

Another take on the interval heuristic is that intuitively it divides the prefix-space (i.e. the subset of all paths in C that begin at v_0) for a given target value into a part with no heuristic guidance and a part with perfect heuristic guidance as figure 3.11 shows. In many problems, an optimal target value path is unlikely to exist (i.e. the valuation of best solution will deviate

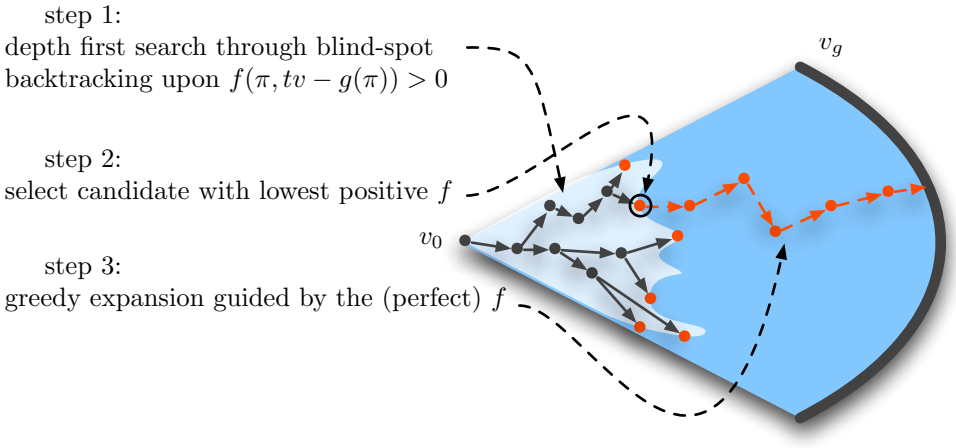


FIGURE 3.12 Basic steps of a DVTVS search.

somewhat from the target value). In this case an admissible search algorithm must *exhaustively* explore (with the exception of redundant candidates and their offspring) the light blue part, but can then "pierce" the dark blue part due to the perfect guidance. Also, redundant candidates are scarce relative to a shortest path search due to the comparatively very restrictive definition of redundancy. At the same time *Open* is costly to represent in memory not only due to the number of elements it comprises but also due to the need to represent each element's complex structure. In summary, for this domain, the costs of duplicate detection in the general case outweigh its savings. These are the guiding insights behind *depth-first target value search* (DFTVS).

The idea is straightforward. First run a depth-first search from v_g which backtracks once the estimator returns a positive value[1]. Before the search, a best candidate reference is initialized to some dummy value $\perp$ with infinite cost. Each generated candidate with a positive estimate (before backtracking) replaces the current reference if its cost estimate is lower. Once the depth-first search finishes, the best candidate can be expanded straightforwardly into the target-value path through the perfect guidance of f (i.e. for a candidate $\pi_{v_0 \to v}$, one appends the successor $v' \in \delta(v)$ such that $f(\pi_{v_0 \to v}, tv) = f(\pi_{v_0 \to v} \circ v', tv)$). Figure 3.12 depicts these steps graphically.

[1] technically one can backtrack when the estimator returns 0 but the derived target value for the suffix *equalled* one of the bounds in the store. For brevity and clarity I omit this implementation detail in the discussion.

Algorithm 7: DFTVS

Depth First Target Value Search

Input: C the connection graph
Input: tv the target value
Input: f an admissible interval based estimator
Data: $\hat{\pi}$ the current best candidate
Output: $\pi^{tv}_{v_0 \to v_g}$ the target value path

$\hat{\pi} \leftarrow \perp$; // assume $f(\perp) = \infty$
if *DFTVS-FB([])* **then**
  return $\hat{\pi}$;
end
return *GEXP($\hat{\pi}$)*;

Algorithm 8: DFTVS-FB

Depth First Target Value Search bounded by f (recursive definition)

Input: $\pi_{v_0 \to v}$ a prefix
Data: C the connection graph
Data: f an admissible interval based estimator
Data: tv the target value
Data: $\hat{\pi}$ the current best candidate
Output: $true$ if $\pi^{tv}_{v_0 \to v_g}$ has been found, $false$ otherwise

if $f(\pi_{v_0 \to v}) > 0$ **then**
  if $f(\pi_{v_0 \to v}) < f(\pi^*)$ **then**
    $\hat{\pi} \leftarrow \pi_{v_0 \to v}$;
    return $false$;
  end
end
if $v = v_g$ **then**
  $\hat{\pi} \leftarrow \pi_{v_0 \to v}$;
  return $true$;
end
for $v' \in \delta(v)$ **do**
  if *DFTVS-FB($\pi_{v_0 \to v} \circ v'$)* **then**
    return $true$;
  end
end
return $false$;

Algorithm 9: GEXP

Greedy prefix expansion guided by f (recursive definition)

Input: $\pi_{v_0 \to v}$ a prefix
Data: C the connection graph
Data: f an admissible interval based estimator
Data: tv the target value
Output: $\pi_{v_0 \to v_g}$ a completion of $\pi_{v_0 \to v}$

if $v = v_g$ **then**
  return $\pi_{v_0 \to v}$;
end
$v' \leftarrow \underset{\delta v}{\operatorname{argmin}}(f(\pi_{v_0 \to v} \circ v'))$;
return *GEXP(*$\pi_{v_0 \to v} \circ v'$*)*;

The f-bounded depth-first search as well as the greedy expansion are most easily defined as recursive algorithms (see listings 8 and 9 respectively). Based on these functions, the actual DFTVS is exceedingly simple (see listing 7). For clarity I assume that the connection graph, cost estimator and (for the depth-first exploration) current best candidate are accessible in the scope of the heuristic functions. The only necessary extension to the above idea is a goal test for candidates generated during the blind search. If DFTVS-FB finds the target value path (i.e. ending in v_g with $f = 0$) it stores the path in $\hat{\pi}$, stops the exploration and signals success to DFTVS by returning $true$. Otherwise it returns $false$, whereupon $\hat{\pi}$ holds the least non-zero cost candidate encountered during the sweep. At this point it is shown that no zero cost solution exists and the best solution is an expansion of the path in $\hat{\pi}$. As f's guidance is optimal, the simple greedy expansion scheme of GEXP will transform $\hat{\pi}$ into $\pi^{tv}_{v_0 \to v_g}$.

3.5 Empirical Evaluation

In this section, I give a comparative evaluation of the modified A^* (HS) with an inadmissible estimator ($f(\pi) = |tv - (g(\pi) + h^*_{sp}(\pi))|$ where $h^*_{sp}(\pi)$ is the lowest valued completion of π or in other words h is a *perfect* shortest-path heuristic) as discussed in section 3.3.1, BFTVS and DFTVS. The algorithms are evaluated using two domain generators: $sparse$ and $dense$. All tests were performed on a machine with a 2.8 GHz Intel Core 2 Duo CPU with 4 GB of RAM running MacOS X 10.5.6. All algorithms are implemented as parts of a uniform framework in C++ to allow fair runtime comparisons.

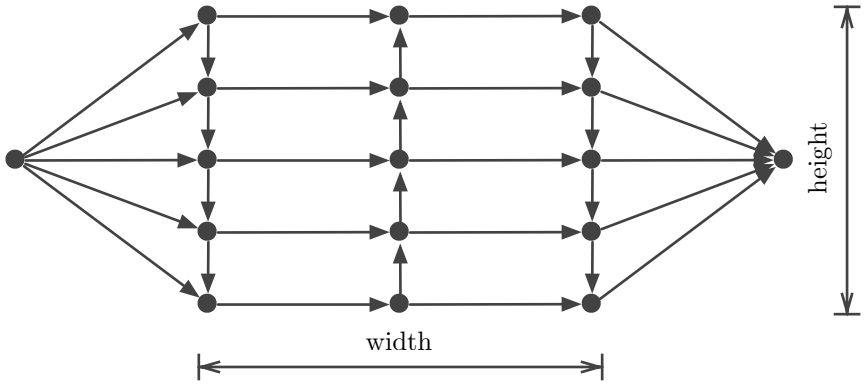


FIGURE 3.13 The *sparse* domain: vertices are always connected to their “right", as well as their “lower" or “upper" neighbors (depending on whether the column is “odd" or “even").

3.5.1 The Test Domains

Both generators produce connection graph lattices, consisting of designated start and goal vertices v_0, v_g and a “grid" of vertices between them. Generally edge values are assigned randomly (sampled from a uniform $(0; 1]$ distribution). Both are parameterized in terms of width, height and a seed value for a random-number generator. In the *sparse* domain, vertices (with the exception of v_0 and v_g) have a constant out-degree of 2, and path lengths (in number of edges) between start and goal vary between width $+ 1$ and height $\times$ width $+ 1$. Its general connection pattern is shown in figure 3.13.

The *dense* domain has uniform path-lengths of width $+ 1$. The generator employs an additional parameter, a probability p, which governs the out-degree of nodes in the grid: a vertex has a connection to a vertex in its “right" neighboring column with probability p (besides its direct right neighbor, with whom it is always connected). The corresponding connection pattern is depicted in figure 3.14. It results in an average out degree of $p(\text{width} - 1) + 1$, (which is approximately $p\sqrt{|V|}$ for the “square" graphs I mostly use in the evaluation). In general, for “square" graphs, I use the term *dimension* (d) to denote width and height parameters. Also, if not otherwise noted, I allow up to 5 intervals per entry (i.e. $k = 5$) and use 0.5 as probability parameter for the *dense* domain.

Both domains are hard in that they contain a very large number of paths between v_0 and v_g (exponential in width with a base of $p(\text{width} - 1) + 1$ for *dense*, and in width $\times$ height with a base of 2 for *sparse*). Note that the ratio of paths to vertices in the example domains given in section 3.1 is usually far lower. Especially for *dense*, the regular structure also allows a straightforward interpretation of which target values result in hard instances . Due to the uniform distributions of edge values, the values for elements of $\prod_{v_0 \to v_g}$ will be normally

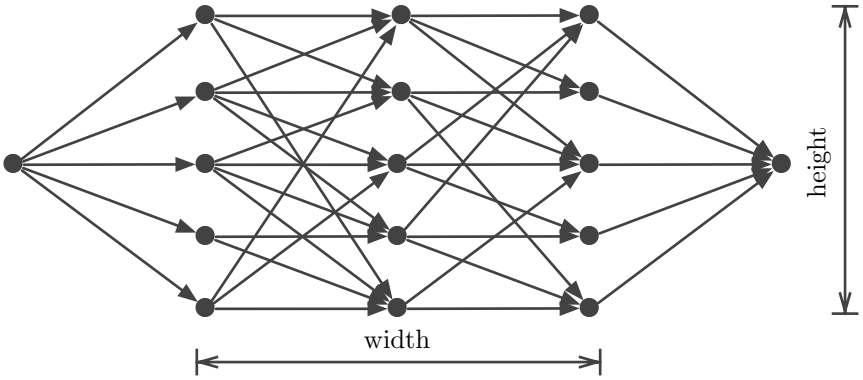


FIGURE 3.14 The *dense* domain: vertices are always connected to their “right" neighbor; additionally, for each “other” vertex in the “right" neighboring column, there is a connection with probability p.

distributed around $0.5(0.5(\text{width} + 1))$ in the interval $(0; \text{width} + 1]$. The closer the target-value to the peak of the normal distribution, the harder the instance as comparatively more candidates are close to this valuation.

3.5.2 Comparison of HS, BFTVS and DFTVS

Figures 3.15, 3.16, 3.17 and 3.18 (note the logarithmic scale of the value axis) give the average query time (in μsec), using the HS, BFTVS and DFTVS algorithms with target-values set to different fraction of the highest valued path in the respective graph. BFTVS and DFTVS query times include database construction, whereas the time for computing shortest-path lengths used by HS' heuristic is omitted (they were retrieved as lower bounds from a pre-computed 1-interval database at runtime). Each datapoint represents an average over 25 graphs specified by different seed values. The runtime distributions reflect the normal distribution of path lengths in the *dense* domain. As expected, the problem is hardest for BFTVS and DFTVS if the target value is half the sum of the lowest and highest valued path of the graph, as then most paths valuations are close to the target value, resulting in a large blind-spot. The relative differences in runtime between BFTVS and DFTVS widen rapidly from one order of magnitude at 6×6 to three orders of magnitude at 8×8, so I limited this comparison to very small graphs (between 25 and 64 vertices, excluding v_0 and v_g). HS fares between another 1 to 3 orders of magnitude worse than BFTVS. Due to its memory overhead, I do not give results for the 8×8 case (incrementing the dimension of *dense* pretty reliably increases the average search time by about two orders of magnitude). In *dense*, HS is consistently hardest for target values around $1/3$ of the longest path's valuation. The algorithm is (unsurprisingly) a bad fit for target value

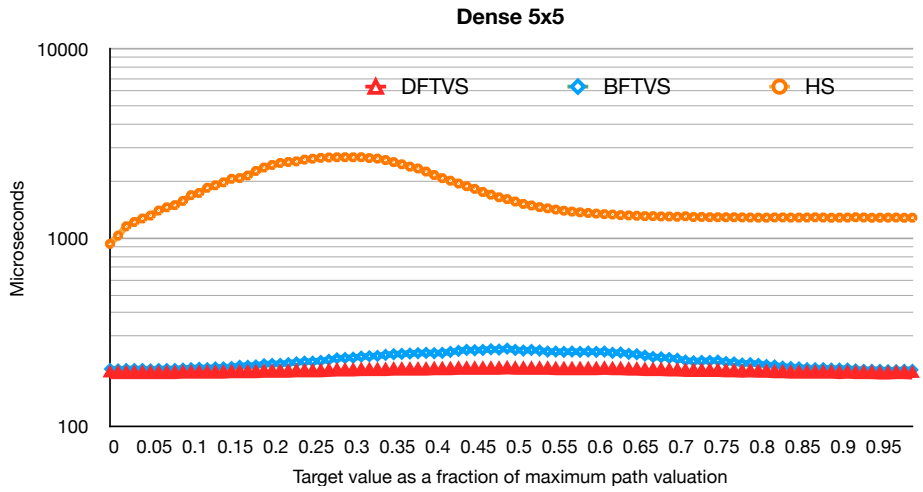


FIGURE 3.15 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 5×5 graphs of the *dense* domain (with different seed values) of HS, BFTVS and DFTVS.

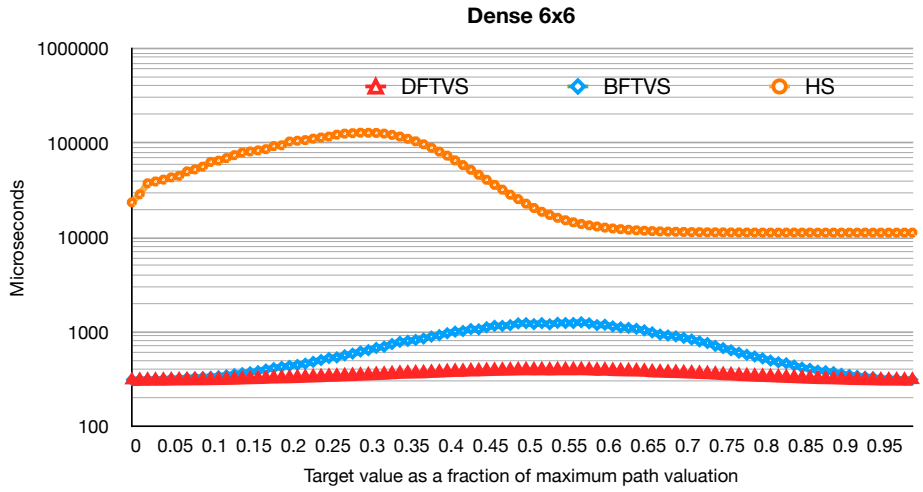


FIGURE 3.16 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 6×6 graphs of the *dense* domain (with different seed values) of HS, BFTVS and DFTVS.

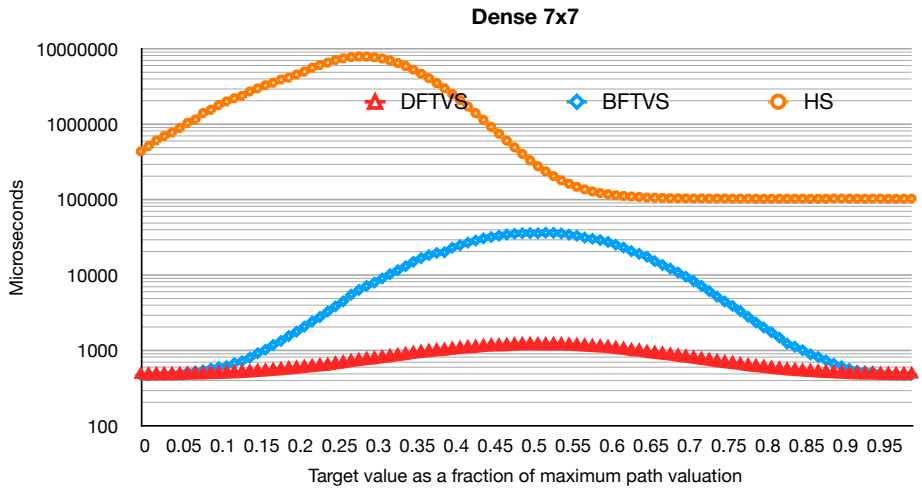


FIGURE 3.17 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 7×7 graphs of the *dense* domain (with different seed values) of HS, BFTVS and DFTVS.

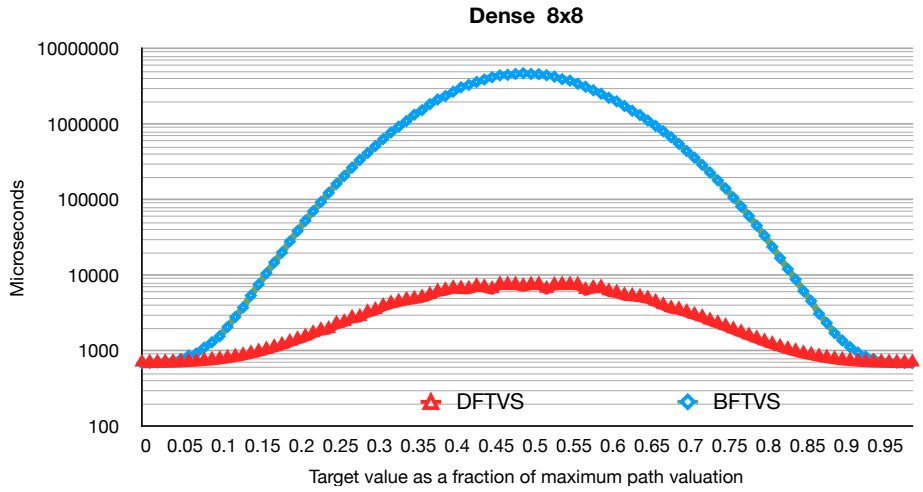


FIGURE 3.18 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 8×8 graphs of the *dense* domain (with different seed values) of BFTVS and DFTVS (HS is omitted due to its poor scaling).

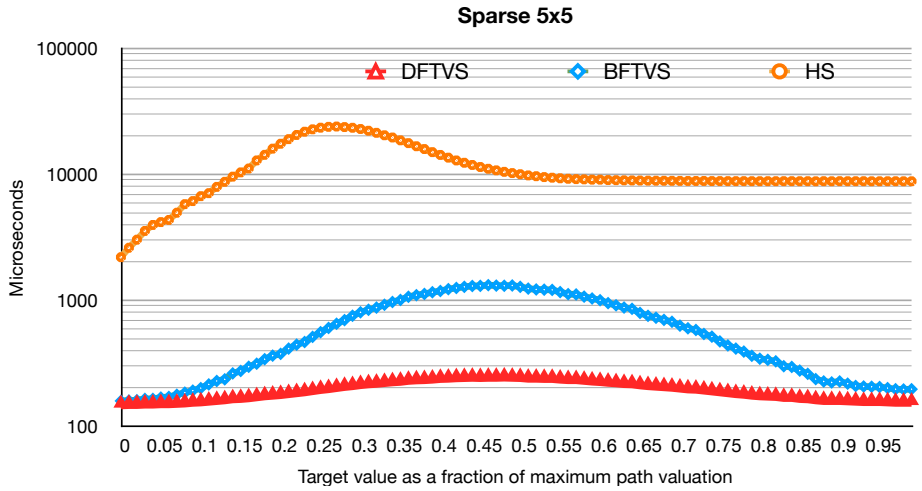


FIGURE 3.19 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 5×5 graphs of the *sparse* domain (with different seed values) of HS, BFTVS and DFTVS.

search. The search must produce multiple solutions to guarantee admissibility and particularly the priority queue proves very costly in comparison to the optimized data structures of BFTVS and DFTVS as can be best seen on the instances with the lowest and highest target values where the problem degenerates into a shortest or respectively longest path search in a directed acyclic graph.

Figures 3.19, 3.20 and 3.21 gives the respective average query time for the *sparse* domain, albeit for smaller graphs (up to $d = 6$ for the HS and $d = 7$ for BFTVS and DFTVS. Due to the larger number of paths in these graphs (see above, i.e. the much higher exponents outweighs the lower base), search times are about an order of magnitude higher in comparison to *dense* graphs with the same number of vertices. The relative results closely mirror those in the *dense* domain.

3.5.3 Scaling of DFTVS

As HS and BFTVS quickly approach time and space barriers, I will concentrate on DFTVS in the following empirical evaluations. Figure 3.22 gives an overview of the runtime distribution of DFTVS queries (μ = blue line, σ = red bars) in relation to graph size. Each datapoint represents 10 instances (differing in their seed values) against each of which 1000 queries were executed with target-values randomly sampled from a uniform distribution between minimal and maximal path valuations in the graph. Average query times show relatively modest growth

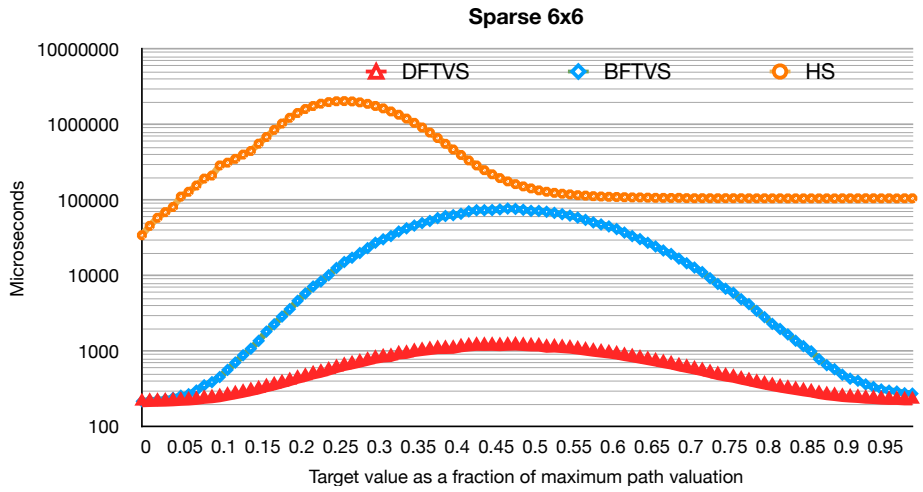


FIGURE 3.20 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 6×6 graphs of the *sparse* domain (with different seed values) of HS, BFTVS and DFTVS.

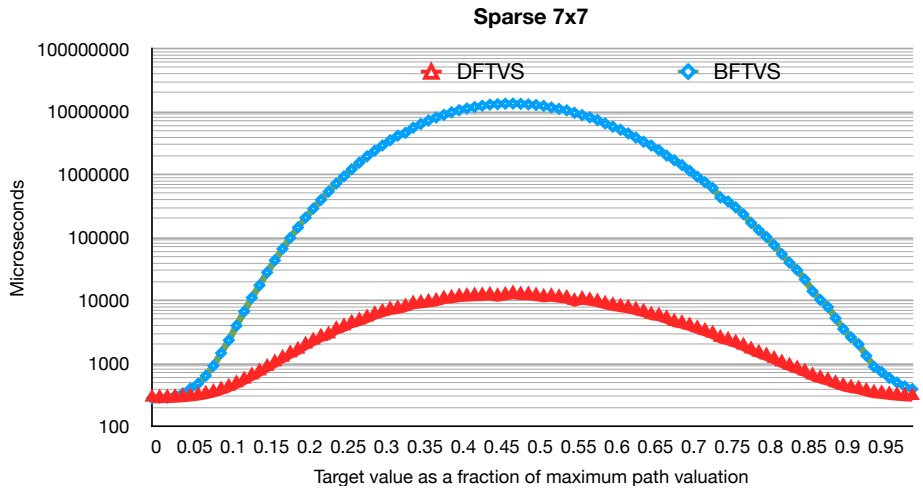


FIGURE 3.21 Average query times for multiple target values (between lowest and highest valued paths in the graph) over 25 7×7 graphs of the *sparse* domain (with different seed values) of BFTVS and DFTVS (HS is omitted due to its poor scaling).

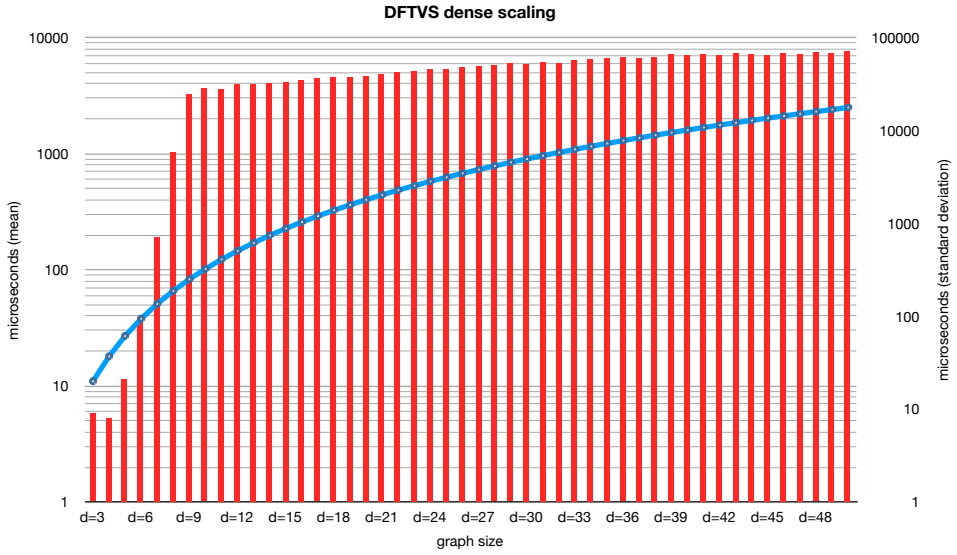


FIGURE 3.22 Mean (blue function) and standard deviation (red bars) of DFTVS query times over uniformly sampled target values in multiple *dense* graphs of different sizes ($d \in [3 \dots 50]$).

in graph size, with their standard deviation following suite largely being around a magnitude larger than the mean. Note how DFTVS's μ, σ for the $d = 50$ graphs (2002 vertices each) are only about $1/50$-th and $1/18$-th of BFTVS's μ ($\sim 9.5 * 10^5$) and σ ($\sim 1.3 * 10^6$) respectively on the $d = 8$ graphs (66 vertices each) from figure 3.16.

Figure 3.23 shows the same for the *sparse* domain on up to $d = 30$ graphs (902 vertices). As *sparse* is a significantly harder domain, only a 100 queries were run per instance (of which again, 10 were generated for each datapoint) for this evaluation. Correspondingly the mean runtime is one to two orders of magnitude higher then for the *dense* instances. The standard deviation of runtime shows a similar pattern as in the dense instances being consistently about one order of magnitude higher than the mean runtime.

3.5.4 Interval Store Evaluation

Figure 3.24 gives average construction times (in μsec) needed to build a 5-interval store for different graphs (*dimensions* 3-90). Each datapoint represents an average over 25 graphs (with differing random seeds). The results show that the overhead for computing the database only represents a small part of total query runtime in both domains. Hence constructing a database even for single queries is hardly an issue from a computational point of view. Note that the graph description size (i.e. $|V| + |E|$) is quadratic in d for *sparse* and cubic for *sparse*

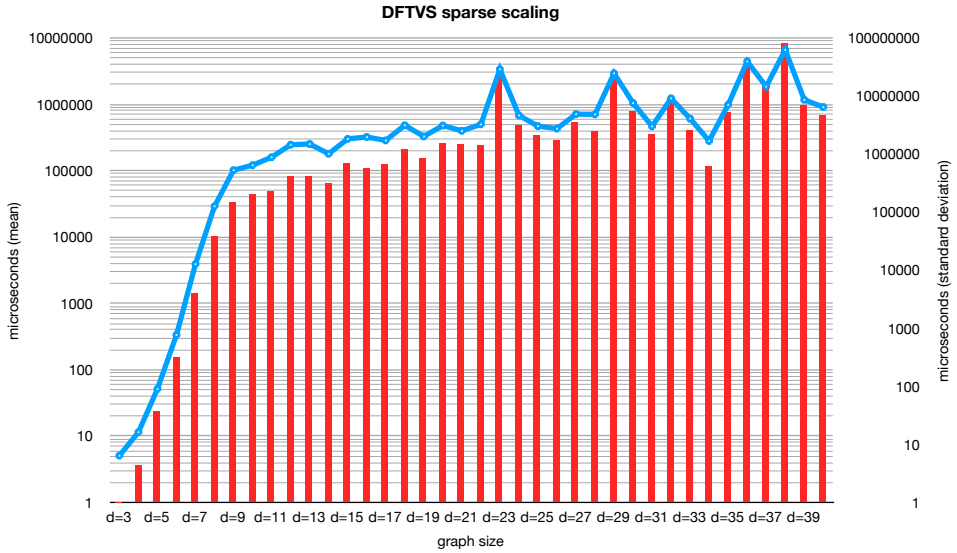


FIGURE 3.23 Mean (blue function) and standard deviation (red bars) of DFTVS query times over uniformly sampled target values in multiple *sparse* graphs of different sizes ($d \in [3 \ldots 40]$).

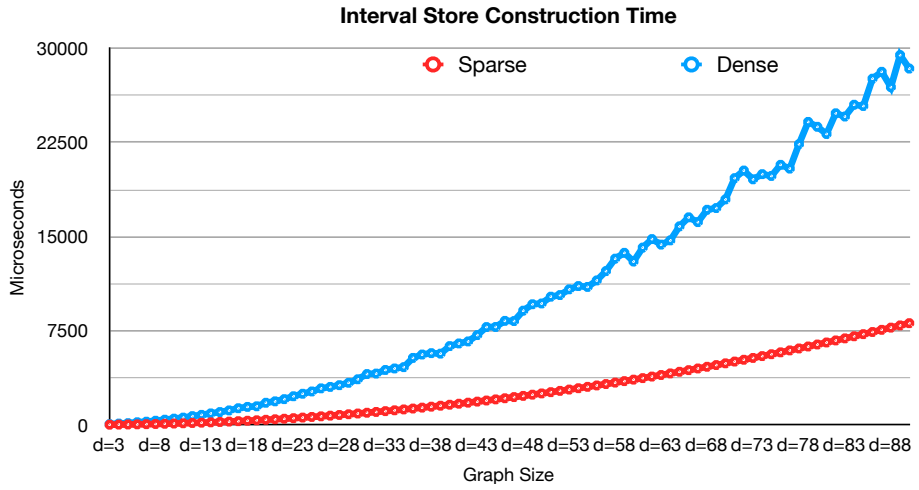


FIGURE 3.24 Average interval store construction times for *sparse* and *dense* graphs of different sizes ($d \in [3 \ldots 90]$, corresponding to graphs of 11 to 8102 vertices).

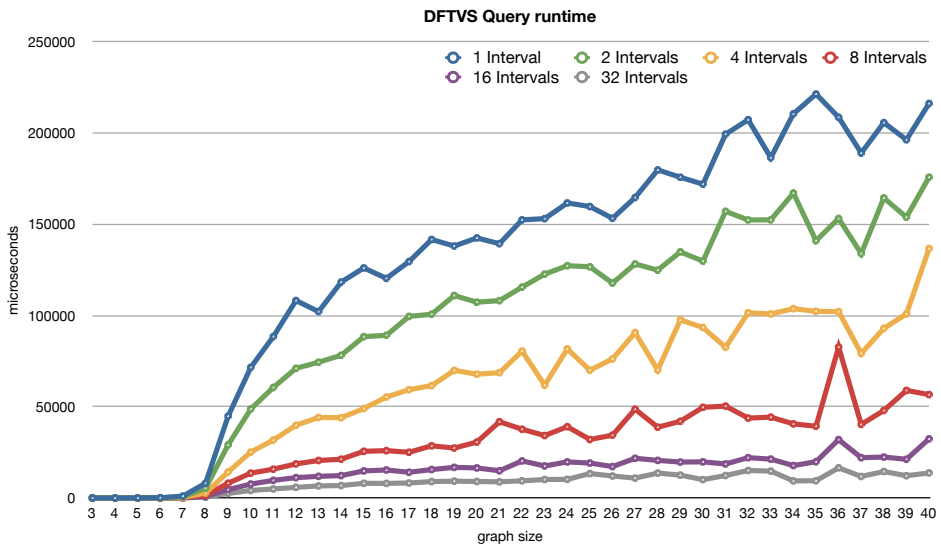


FIGURE 3.25 Average query time for $dense$ graphs of different sizes ($d \in [3 \dots 40]$) in relation to the number of intervals per entry.

as the average number of edges is $2|V|$ in the former and $p(d-1)|V|$ in the latter domain. The resulting larger computation time for building the store is the only sense in which $dense$ is the harder domain. For all other purposes, the much larger number of paths between v_0 and v_g in the $sparse$ domain make it a much harder search problem.

Lastly, figure 3.25 shows how the number of intervals per entry (i.e. constant k) influence search time. For this evaluation, I generated 5 $dense$ graphs for each dimension (from $d = 3$ to $d = 40$ corresponding to 11 and 1602 vertices respectively). For each graph 200 target values were randomly (uniformly) chosen between the lowest and highest path valuations occurring in the respective graph. Corresponding queries were then executed with the different databases and their runtime averaged. With the exception of single and dual intervals per entry, doubling the number of intervals roughly halves the query's runtime. This coincides quite well with the observations of Holte [HH99] about the relation between size and accuracy of memory-based heuristics.

3.6 Summary

In this chapter I have introduced a novel class of combinatorial search problems, an efficient memoization technique that allows for deriving *admissible* estimates for the subdomain of a-cyclical connection graphs and a corresponding *domain-dependent* algorithm that exploits properties of these estimates to drastically improve scalability. This also serves as a good

example of the innate challenges of domain independent planning.

For one, there are no *silver bullets*. Techniques such as duplicate detection are a big benefit (and sometimes even a requirement) when tackling large instances in many domains but there are nearly always *relevant* problems where they prove highly detrimental. One potential strategy to deal with such issues is to design domain independent planners as sets of interchangeable search, estimation and storage policies that are combined into concrete planning algorithms at runtime based on domain and instance analysis of some top level component. State of the art academic domain independent planners feature this to a limited extent in e.g. allowing the user to select between different heuristics or modify the search algorithm slightly. The popular SPIN explicit state model checking tool [Hol97] is a good example of such a design philosophy.

Another is that domain knowledge is very valuable. It is not atypical that the exploitation of even small insights such as the nature of the above estimates reduces the necessary effort by orders of magnitude. In other words, domain-dependent planning will always be relevant in challenging domains. It however likewise strongly depends on the availability of comprehensive toolkits of interoperable search technologies and corresponding software components that can be easily adapted to economically transform domain know-how into efficient planners.

CHAPTER 4

State-set Representation

I will now turn to state and state-set representation, a topic that I have only touched on in very coarse terms so far. The empirical results of section 3.5 already gave a hint of its criticality. For planning, efficient management of search states lies at the heart of memoization techniques and in particular unit-cost best-first algorithms. The scope of this chapter will hence be more general than the last. The time and space complexity of handling large state sets and maps, be it in the form of redundant states for duplicate detection, search-frontiers to guarantee best-first properties or as memory-based heuristics, profoundly influences the solvability of search problems. Of special importance for any reasonably general search algorithm is that the employed representations strike a good balance between small size and efficient operations.

4.1 Background

First, I want to provide some background and coarsely discuss where and what the challenges associated with state-set representation are. While the techniques I introduce later in this chapter are fairly general, I limit the following discussion to the context of propositional planning as introduced in chapter 2.

4.1.1 Pattern Databases

Before delving into representation techniques I want to shortly discuss Pattern Databases [CS98], an important class of memoization heuristics. Conceptually pattern databases are enumerations of all possible subgoals (up to a certain size) with associated costs (i.e. f^*) for their optimal solutions. The subproblem instances are represented by patterns and are stored in a map such that given any concrete state, it can be efficiently matched to its pattern and the corresponding value can be retrieved (c.f. the interval store of chapter 3). In propositional

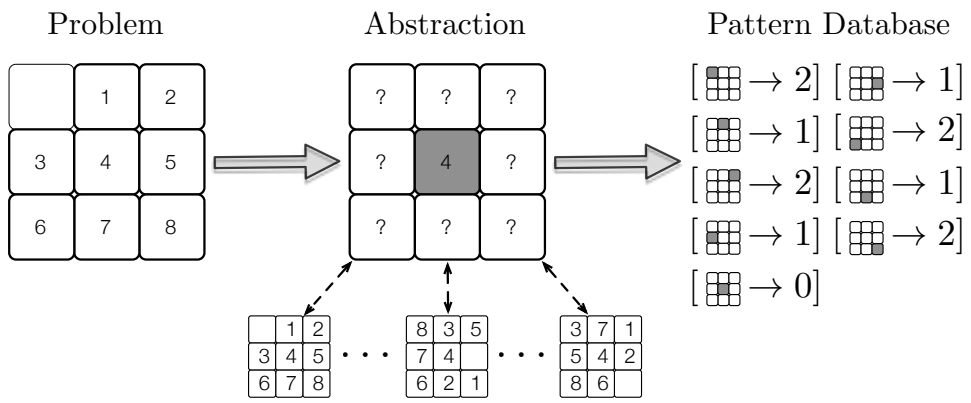


FIGURE 4.1 The 8-puzzle domain, its abstraction to tile 4 and the corresponding pattern database. There is one entry in the database for each each possible problem instance in the abstraction. Each entry comprises of a description of the problem instance and the minimum cost of solving it. These abstract problem instances are referred to as "patterns" as they each correspond to multiple concrete problem instances.

planning these subgoals stem from a derived, (relatively) easy-to-solve domain that is usually defined by an abstraction of the original problem. A suitable abstraction must be interpretable as a (usually not invertible) function mapping original problem configurations to their abstract counterparts. These abstract problem configurations are referred to as patterns. One then solves the abstract problem for all patterns and stores them with the associated costs of their optimal solutions in a database. A practical example is given in Figure 4.1. In this example the abstraction removes the identities of all tiles but tile 4. By associating each configuration of the 8-puzzle to the pattern with the matching position of tile 4, this can be interpreted as a many-to-one mapping function. Depicted on the right of Figure 4.1 then is the resulting pattern database. From a theoretical perspective, pattern databases (or PDBs in short) define one of the four fundamental classes of heuristics known for propositional planning [HD09]. For the purpose of this chapter they provide a good use-case of memoization techniques in propositional planning due to their conceptual simplicity and practical relevance (see [CS96], [Kor97] [Ede02] and [KF07] amongst many others).

4.1.2 State Representation

I will begin the discussion of representation techniques on the level of state encodings. To this end, I return to propositional planning and the example problem as given in chapter 2. The intuitive way to represent states of a propositional planning problem is as an array of assignments covering the propositional variables. Each position corresponds to one variable.

Nearly all programming environments offer support for some binary type such as $bool$ in C++ language. In the apartment domain, a corresponding state comprises an 8 $bool$ array. The first thing to note is that in many domains, non trivial subsets of the propositional variables are mutually exclusive in their assignments. In the apartment domain, this is the case for variables $p_A \ldots p_D$ which represent the location of the agent. Intuitively only one of them can be $true$ in any sensible state. Hence on a logical level, out of the 16 possible assignments to $p_A \ldots p_D$ only four really occur. From an *information theory* perspective the maximum entropy of these four propositional variables is only $\log_2 4 = 2$ Bit. Domain properties of that ilk provide ample opportunity for representational simplification and corresponding *multi-valued* representations (such as the popular *simplified action structures* SAS [SR86] and SAS+ [BK91] formalisms) for propositional planning are widely used in planning. Here such groups of mutually exclusive propositional variables are swapped for corresponding multi-valued variables, e.g. $p_{Room} \in \{A, B, C, D\}$. Note that these formalisms offer the same expressiveness as propositional planning [BN95]. Robust automated domain translation algorithms which extract multi-valued variables exist (see [EH00] and [Hel06a]). These transformations are also beneficial from a computational standpoint. For practical purposes, manipulating an integer is as costly as a boolean, so in the apartment example a room transition reduces from updating two memory values to a single one.

This relates to a second optimization opportunity. Modern processors operate on the level of machine words. For example C++ compilers usually map $bool$ values to 8-bit integers or when aggressively optimizing even to 32-bit integers[1]. Under these conditions a state in the simple apartment domain can occupy between 40 (multi-valued) and 256 Bit (single valued, o3). This extreme runtime bias is often problematic for best-first search due to the large number of states that need to be held in memory. A simple and popular workaround in search and planning are *packed states*. Here, multiple state variables are stored in a single machine word using basic bitwise and arithmetic operators. This is a technique with little computational overhead for reducing storage requirements. For example, in many problems, most variable domains rarely exceed 16 different assignments and hence the variable is representable with only 4 Bits. For such cases, a packed representation can reduce the average storage requirements by a factor of about eight (as opposed to using a full 32-bit word to store each variable assignment).

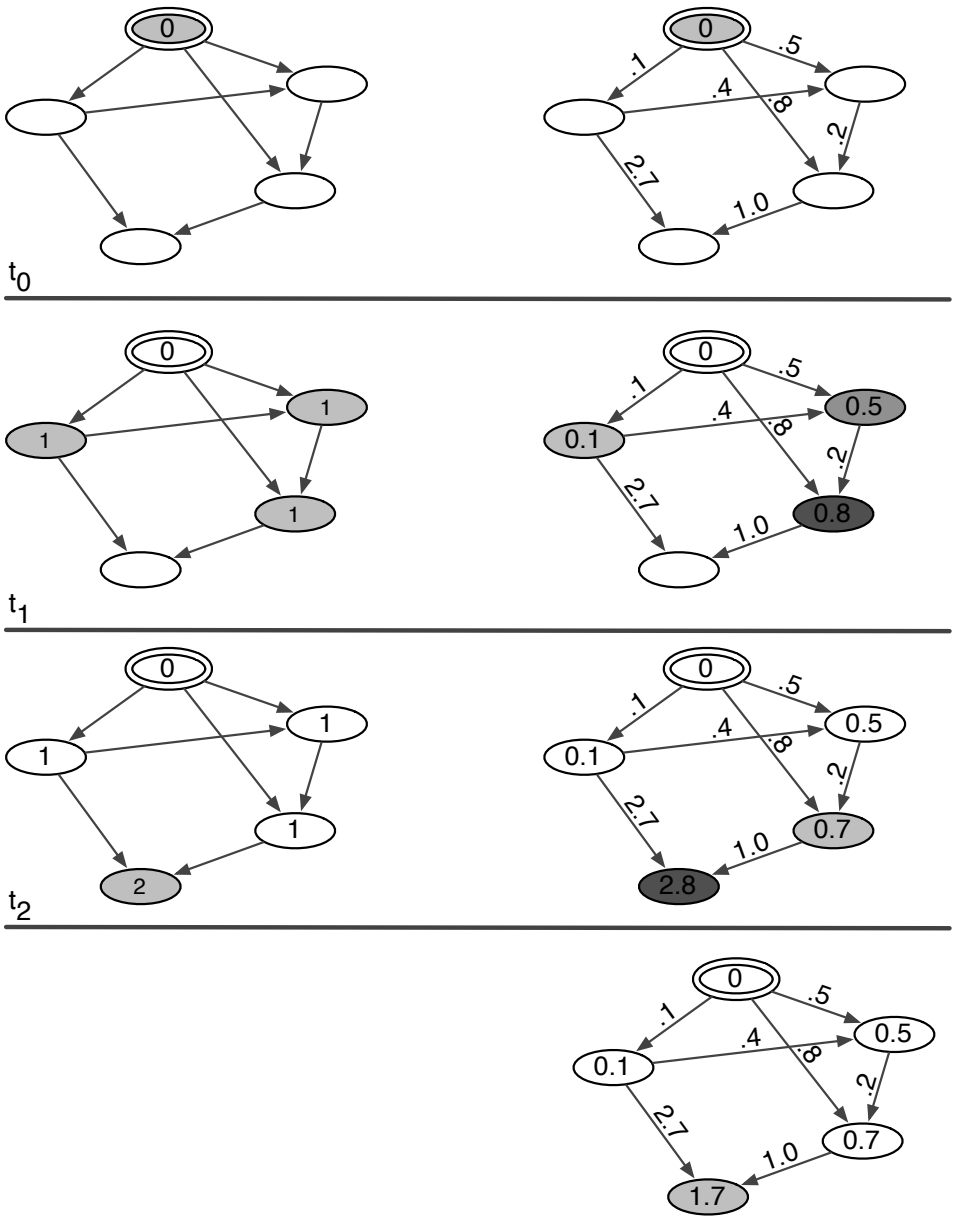


FIGURE 4.2 Dijkstra's algorithm with unit-(left) and variable-(right) edge-costs over a small explicit graph. *Open* at each time step is denoted by the gray nodes with the frontier comprising of the nodes with the lightest shade. Values in the nodes are the currently ascribed g-values.

4.1.3 State-Sets in Unit-Cost Best-First Search

During planning, states usually occur in and are manipulated as part of groups . Linear-space search algorithms such as IDA* [Kor85] are specfically designed to limit the size of occurring state collections and hence use much less memory than best first algorithms such as A* but by forgoing duplicate detection and global expansion orders they pay the price of extra node expansions to find optimal solutions. This time-space tradeoff pays off in domains with few duplicates such as the sliding-tile puzzles or target value search where IDA* and DFTVS easily outperform A*, but many domains (e.g. multiple sequence alignment) are not conducive to this approach. Hence current state-of-the-art heuristic search planners such as Fast Downward [Hel06a], HSP$^*_F$ and GAMER [EK08a] include full duplicate detection in their search algorithms. The nature of best-first search is to expand candidates in ascending order of their f-estimates. Guaranteeing this constraint requires buffering newly generated candidates in $Open$. In the general case (i.e. real-valued estimates and inconsistent heuristics), such a buffer must support efficient retrieval/removal of the candidate with the lowest estimate (i.e. highest priority), addition and removal of candidates, tests whether a particular candidate is present in the buffer (set-member test) and changes to candidates' f-values (i.e. priorities). One of the numerous priority queue implementations such as Binomial-Heaps[Vui78] or Fibonacci-Heaps[FT87] are usually backing $Open$ in general best-first search implementations.

In contrast to variable-cost Best-First-Search, $Open$ in unit-cost best-first search usually breaks down to small number of f-value *layers*. These few different f-values, each associated with a relatively large number of states can be much more efficiently represented as a list of sets, lists or deques. Figure 4.2 shows this with a small example of Dijkstra's algorithm[Dij59] (basically A^* with a Null-heuristic). Note that for the special case of a null-heuristic the f-estimate reduces to the g-value (i.e. the distance from the initial state) and hence $Open$ to an equivalent, flat set (i.e. $Open$ is equivalent to the search frontier).

To enable duplicate detection, a search algorithm must keep track of all *unique* states it has generated at any point in time. To this end it usually maintains $Close$, a set comprising of encountered states not in $Open$.

Now I will give a quick overview of the most critical operations performed on sets in unit-cost search.

[1] for example GCC with an o3 flag on the OS-X platform

4.1.3.1 Set member tests and member associated data

Generated candidates are tested against $Close$ and $Open$ for duplicate detection. Furthermore, many search algorithms rely on the association of meta-data with known states, such as f-values and predecessor identifiers. When using pattern databases, the corresponding abstraction of the state is also tested against the database to retrieve the associated h-value. Note that even with duplicate detection one and the same state can be (and generally is) generated multiple times during a search. Set-member tests and retrieval of meta-data are hence generally the *most frequent* set operations in search.

4.1.3.2 Set iteration

In unit-cost BFS candidates are expanded in logical phases. Each such phase consists of generating the successors of all candidates in the lowest (f-value) layer in $Open$ (i.e. the search frontier). Overall this set iteration is somewhat less critical than member tests, as theoretically only non-duplicate states are iterated over once during a unit-cost search (if the heuristic is consistent).

4.1.3.3 Set creation and union

Nodes generated during a layer expansion which passed duplicate testing need to be inserted into the appropriate $Open$ layer for later expansion. Meanwhile, after a state has been expanded, it needs to be removed from $Open$ and inserted into $Close$. A common optimization (if the representation supports it efficiently) is to merge the frontier *in toto* with $Close$ at the end of a layer expansion instead of moving individual candidates during the expansion.

4.1.4 Set-representation techniques

Commonly used data structures for state-sets in search can be classified into *explicit* and *implicit* representations. I classify as the former any structure whose constituents' representation is some function of the individual states, i.e. the set-representation of a state is only dependent on the assigned state itself. The latter usually comprise some information derived from the states, the set composition and/or the problem space from which its individual elements can be reconstructed on demand. In the following, I will present examples of set representations for state-sets and maps as found in state of the art propositional planners.

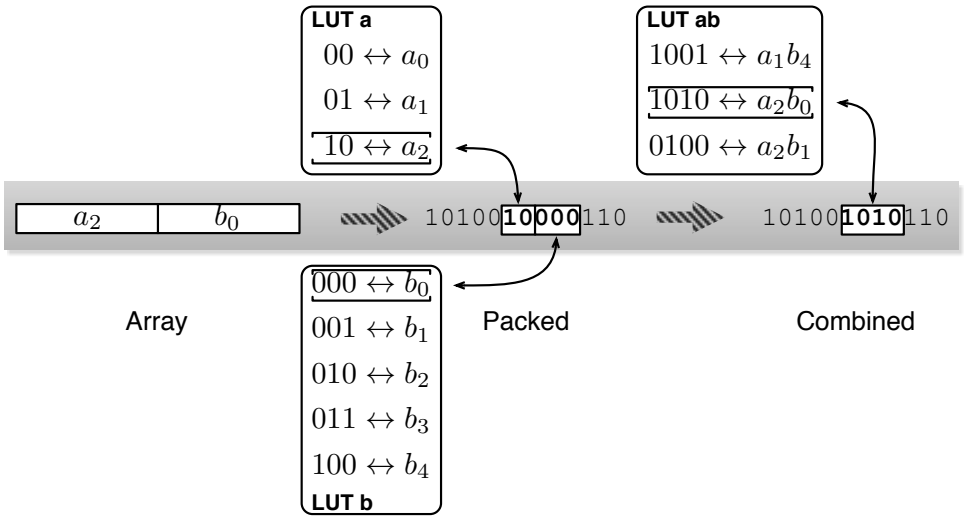


FIGURE 4.3 State variable assignments a_2 and b_0 in Array, Packed and Combined representations of a state, requiring 2 Words, 5 Bits and 4 Bits of memory

4.1.5 Explicit set representations

As mentioned above, states of discrete planning problems have a more or less "natural" machine representation as arrays of multi-valued assignments. The most widely used explicit set representations are based on standard container library primitives. Pointers or entire value arrays are organized in fairly standard dynamic data structures such as sorted lists, deques (backed by dynamic arrays or single or double-linked lists), sets (most often implemented as red-black trees [Bay72]) or hash-tables (variants of extensible hashing [FNPS79]). Fast Downward [Hel06a] is an example for popular state-of-the-art planner using STL vectors and maps for representation.

A straightforward extension is to represent individual elements by compressed derivatives of their "natural" representation (i.e. an array of state variable assignments) such as the packed representations sketched above. Forming the cross-product of state variable domains is a way to further increase coding efficiency. For example, two domains with respective sizes 3 and 5 would individually require 2 and 3 Bits to encode for a total of 5 Bits. Forming the crossproduct yields 15 different assignments which can be encoded in 4 Bits. Ultimately, by using the crossproduct of all fluent domains, one ends up with an enumeration of all possible state assignments. As long as the domains of combined variables remain relatively small, lookup tables (or short LUTs) can be employed for quick mapping between natural and packed or combined representation. These LUTs grow with the crossproduct's domain. To maintain a net reduction in storage size runtime can be sacrificed by using the original, small fluent-

level LUTs and computationally projecting the combined fluent to its original constituents during (de-)coding. This makes for a good example of the runtime and space trade-off that will permeate this chapter. Explicit representations map more or less directly to high performance container libraries readily available for basically any commercial-grade programming language. As, from a best-first search point of view, the design of most of these components is strongly biased to favor runtime over size, they are usually only competitive on large problems in planners relying on complex and computationally expansive heuristics where the number of generated states per unit of runtime is comparatively small (hence those planners usually run out of time before saturating available memory).

4.1.6 Implicit Set Representations

4.1.6.1 Statemaps

The first implicit representation is a straightforward extension of the above enumeration thread. The crossproduct over all state variables can be interpreted as a minimal perfect hash function for states. That is it bijectively maps each state to a unique integer in range $[0; |2^P|)$, making a representation of the actual state in the hash-table superfluous. For propositional planning simply interpreting the packed assignment vector as a binary encoded integer produces such a mapping. All that needs to be represented is the information associated with states (e.g. a presence bit, distance to the initial node, etc.). Such hash tables have a size that is a constant factor of the state-space's cardinality. While generally very time efficient, state-maps are only space efficient or in fact reasonable if one expects first that the state space is sufficiently small and second that the such represented set ultimately comprises a significant subset of the total state space. In domain independent planning this is reasonable to assume only for small and highly abstract pattern databases where usually most of the possible variable assignments are reachable and thus present in the set. In domain dependent planning, with hand-crafted enumeration functions that allow to skip a large share of unreachable states, such representations have also been used with very good success for large pattern databases (e.g. for the sliding-tile-puzzles [KF02] or tower-of-hanoi domains [KF07]). In all its properties relegate the technique to more or less special cases.

4.1.6.2 Binary Decision Diagrams

A more widely applicable variation of the idea of representing state-sets as functions that map elements of the state-space to a binary co-domain denoting membership is based on Binary Decision Diagrams (BDD). BDDs are used to represent boolean functions in memory. A flurry

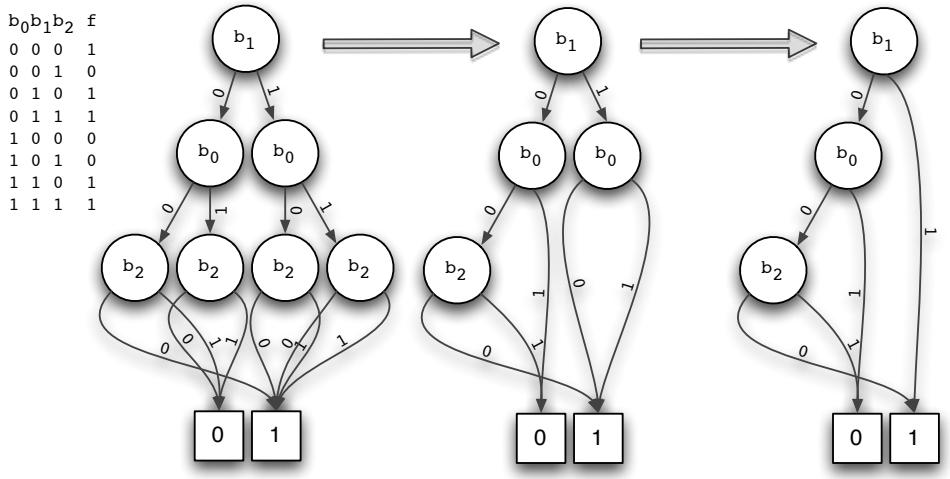


FIGURE 4.4 BDD for f with order $b_1 b_0 b_2$. The example shows how the exponential (in the number of arguments) function table is transformed into an ordered and reduced BDD through repeated pruning of nodes with isomorphic children.

of variants and acronyms exists in the literature, but when henceforth BDDs are mentioned, I refer to ordered and reduced binary decision diagrams. BDDs were originally introduced by CY Lee[Lee59] in the late 1950s, but their full potential as an efficient general data structure was only realized by Randal Bryant[Bry86] when he imposed a fixed variable ordering on the decision diagram, creating a canonical representation for boolean functions. This normal-form allows to map many logical operations on boolean functions, such as conjunctions, disjunctions, negations and abstractions to polynomial-time graph manipulation algorithms. BDDs represent functions as rooted, directed acyclic graphs consisting of decision nodes and terminating nodes true and false. Each decision (inner) node corresponds to a variable of the function and sports exactly two successors for assignments true and false. The variable-order governs the order of variable appearance in every path from a root to one of the terminating nodes in the graph. Two reduction rules govern the minimization of these graphs. The first is to merge all isomorphic subgraphs and the second is to remove any node from the graph that has two isomorphic children.The order together with the repeated application of the two basic transformation rules leads to the compressed and *canonical* function representation of ordered and reduced binary decision diagrams (see figure 4.4 for an example).

The promise of BDDs lies in their potential to represent and manipulate functions over exponential domains in polynomial time and space. In the worst case a BDD's graph-size is exponential in the number of the represented function's arguments (i.e. is a linear function of the size of the domain). In practice, their efficiency is highly dependent on the selected

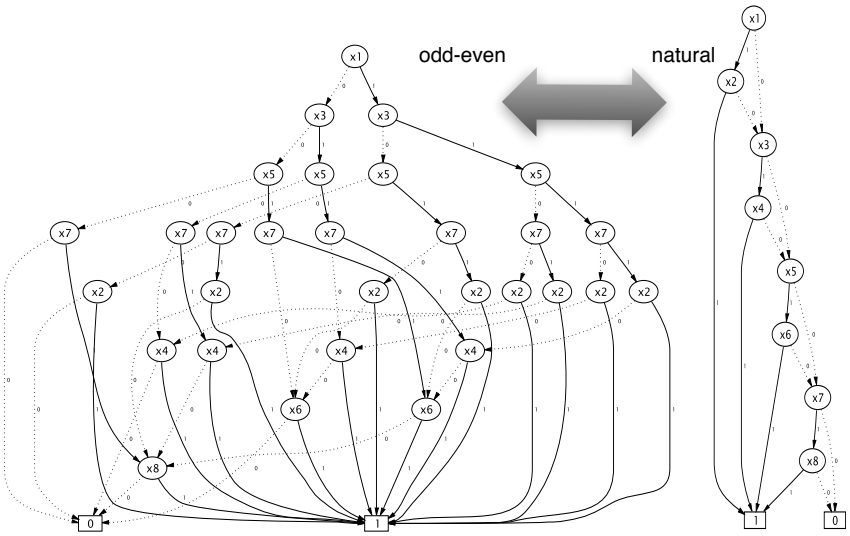


FIGURE 4.5 Ordered and reduced binary decision diagrams for $f(x_1, ...x_{2m}) = x_1x_2 + x_3x_4 + \ldots + x_{2m-1}x_{2m}$ with odd before even variables on the left and natural order on the right resulting in graphs of exponential (2^{m+1}) and linear ($2m$) order respectively (based on diagrams by Dirk Beyer, 1995 used under GFDL).

variable ordering. For an example of how a bad order can increase the size of the graph by orders of magnitude, see figure 4.5. Unfortunately, finding the optimal variable ordering for a given set of functions has been shown to be an NP-hard problem(see [BW02]). Usually initial orderings are determined using either domain knowledge or (more or less) informed guesses. Refined implementations (i.e. BuDDy [LN99] and CUDD [Som97]) allow to dynamically adapt the variable order by exploring permutations derived from greedy heuristics or simulated annealing and reorganize their internal graph representation correspondingly (see, among others [ISY02] and [PSP94]).

Search techniques based on BDDs were first developed in the field of symbolic model checking[McM93] and later transitioned to lifted planning [CGGT97]. They were expanded to directed-search [ER98], [EH01] and later developed to somewhat integrate strong heuristics [JBV02] (i.e. heuristics with a comparatively large range of occurring h-values strongly dependent on individual variable assignments). It is reasonable to assume that for most classes of planning problems, any problem state can be represented as an assignment to a binary variables $x_0, \ldots, x_{m-1}$ (See the above section about packed representations for propositional planning as an example). By mapping such assignments to $true$ or $false$, a set-interpretation of such a boolean function is straightforward.

Empirical analysis of their compression power [BH08] show that BDDs work remarkably well in some domains. As the representation allows efficient manipulations on the function (i.e. set) level, BDDs more or less lend themselves to lifted planning approaches. The basic idea of lifted, breadth-first search is to start with BDD representations of the successor function δ and the initial frontier (i.e. a singleton set comprising the initial state) $l_0 = \{i\}$ of a problem. Layer expansions can then be computed as the *composition* of δ and the layer, i.e. $l_{n+1} = \delta \circ l_n$. Goal checking can also be done on the function level by an existential abstraction of g (the BDD encoding the goal state-set) and a layer, i.e. evaluating the predicate $\exists s : g(s) \wedge l(s)$ where s is some complete variable assignment.

Their spatial and computational efficiency depends on the size of the underlying graph. As even space-optimized implementations specifically targeted at model-checking require between 16 to 20 bytes to represent a node, a BDD can quickly require orders of magnitude more storage than an equivalent packed representation if the normalization process fails to uncover lots of redundancies in the graph. Note the strong dependence on the chosen variable order. Finding good orderings for arbitrary SAS+ instances/domains is more or less an open problem. Work done so far has mostly focused on exploiting static domain properties captured in the variable's causal graphs (see [EH01]). A particular problem with exploiting dynamic (i.e. set-specific) properties through dynamic reordering is that this order pertains to

all represented sets and the quality of such an order can hence deteriorate quickly as more sets are created during a search. This can lead to large spikes in peak memory requirement before re-orderings can bring the size back in check and significant computational overhead as reordering a large BDD is a relatively expensive operation and dynamic optimization techniques usually require multiple iterations to find a good order. A more severe problem is the representation's dependence on problem structure. A particularly challenging case is when state variable assignments in a set represent permutations, in which case Hung has shown that the number of nodes in a BDD is always a function of the number of represented states, regardless of the variable ordering [Hun97]. This happens for example when a task includes some assignment (sub)problem (i.e. of some mutually exclusive resource to different entities, or the position of the agent in the apartment domain of chapter 2) which is quite common in classical planning. Pathologic examples of assignment problems are the sliding-tile puzzles (see also[EK08b]). Another barrier for employing BDDs with many search algorithms is that there is no efficient way of associating data with individual states. The usual workaround is inverse association, where sets are divided into disjunct subsets corresponding to the values of associated data. This scheme works well as long as these subsets remain large or in other words there is little variance in the associated data. For high variance data this is usually not workable as the resulting large number of small sets (i.e. BDDs) offer little opportunity for graph compression along with high house-keeping overhead.

In all BDDs, when coupled with suitable search algorithms, are a powerful representation technique in amendable domains. However those only represent a faction of relevant problems. For example, while the domains chosen for the optimal sequential track at the last international planning competition were by and large favorable to BDD compression as demonstrated by GAMER's [EK08a] overall performance(see [HDR08] for the detailed results), it failed to solve even the smallest instances in domains (e.g. parc-printer) that deviated from that trend.

4.2 LOES - the Level-Ordered Edge Sequence

In this section, I will describe the Level-Ordered Edge Sequence (LOES) a technique for space-efficient set representation in state-space search. The main motivation for its development was to address the widely varying space and time performance of BDDs in many domains. For qualitative and quantitative performance comparisons, I chose packed representation as a baseline, as it is both popular in practice and its sizes scale linearly in the number of represented states and their complexity. As the later evaluation will show relative performance of BDDs over packed representations ranges from exceptional (i.e. unguided search in grip-

per) to dismal (guided search in the n-puzzles) based on structural properties of the underlying domain.

Additionally, as mentioned above, BDDs are problematic when represented sets are small or ancillary information for elements must be stored. A practical example for the former case is best-first search in conjunction with pattern databases. As their (potentially) space-efficient representation in BDDs enables one to employ more informative (i.e. usually larger and less abstract) PDBs, the cardinality of h-values in the database necessarily increases. To avoid re-opening nodes, most unit cost search algorithms expand equal f-estimate candidates in ascending order of their g-values. Ceteris paribus, this leads to fragmentation of the frontier into large numbers of equally promising (and individually small) state-sets, accentuating the weaknesses of BDDs as associative structures through the corresponding decrease in space efficiency. The work done on automatic fragmentation (see [JBV02]) omits discussing space efficiency (probably for this reason) and reports sizable gains in time efficiency only in comparison to undirected BDD search on a very small subset of the structurally well-behaved BDD domains. Likewise, there is little evidence supporting the restriction to weak heuristics or undirected search in order to employ BDDs is *universally* worthwhile from a performance viewpoint. Domain independent planners (and architects of a domain dependent planners) benefit from a representation (at least as a fallback) that is reasonably robust to structural domain properties.

LOES aims to combine the robustness of proven techniques such as packed representations with the benefits of redundancy elimination across set members. Its design goal was to offer good spatial efficiency over a wide amount of domains with a strong worst-case guarantee. Usual operations need to be supported in a time and space efficient way. Of particular practical concern are their peak memory requirements. To enable efficient directed search, association of data to individual set elements should be possible with little time and space overhead.

The idea is to logically represent state-sets as prefix trees (analogous to the compressed graph representation of BDDs), provide a space-efficient encoding for these trees which allows time efficient execution of the required operations on the compressed structure and a way to efficiently adapt these trees as new elements are inserted during a search. In comparison to the less constrained BDD graphs, LOES only allows for a lesser degree of exploitation of redundancies amongst set members -namely commonalities and correlations of elements' state variable assignments- however the more regular nature of the resulting graphs is amenable to a much more efficient in-memory representation. Like BDDs all set operations can execute directly on the compressed structure. Unlike BDDs all operations sport very little memory overhead. In contrast to BDDs, LOES allows economic association of arbitrary data records

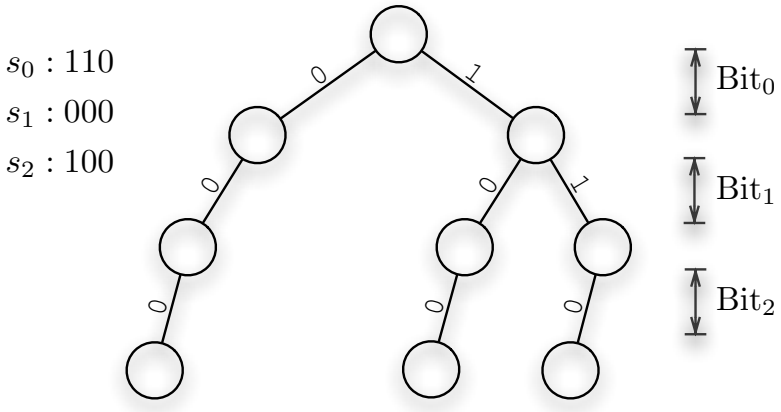


FIGURE 4.6 Prefix tree for a set of three states.

with individual states, i.e. it can be used as a *map* or *dictionary*.

4.2.1 Conventions

First, some necessary conventions and definitions. A reasonable assumption holding in most planning formalisms is that the encoding size of a state of a search problem can be determined upfront (e.g., before the start of the search). For example in pSTRIPS, states can be represented as fixed-size bit vectors, where each bit represents the assignment to one conditional variable of the problem instance. For SAS+ or other multi-valued formalisms, array, packed or combined representations (as are shown in figure 4.3 above) can be used to represent world states as fixed-length bit-strings. Without loss of generality, I assume that any state for a given problem can be encoded in m bits. In the following I will refer to these state encodings as bit-strings or simply strings. Any set of such states (i.e. their strings) can be mapped into an edge-labeled binary tree of depth m with labels $true$ and $false$ like the example set in figure 4.6. Each level of the tree represents a bit position of the member strings. Edges from a node at level i denote $true$ or $false$ assignments to the bit at position i in the string. Every path from the root to a leaf in such a binary tree corresponds to a unique state within the set and can be reconstructed by the respective sequence of edge-labels. The mapping is hence bijective. In this context, I will refer to this logical interpretation of a set of strings as its prefix tree because all elements represented by a subtree rooted by some inner node at level i share a common prefix $b_0 \dots b_i$ in their string representation denoted by the path from the root to that node.

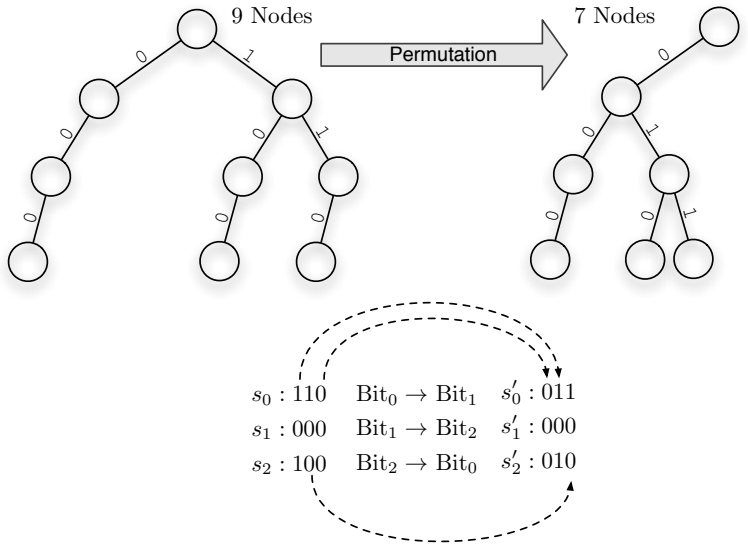


FIGURE 4.7 Permuting the encoding's bit-order can reduce the size of a prefix tree.

4.2.2 Prefix Tree minimization

The order (and hence storage efficiency) of a prefix-tree graph depends on the average length of common prefixes shared between elements of the represented set. This in turn can be influenced by reordering the bits of the state-encoding (see Fig.4.7 for an example). One way to find a suitable order efficiently is a greedy search through the permutation space of the state encoding maximizing the average length of the prefixes amongst set elements at each step.

4.2.3 Sampling representative states

As the permutation space is quite large (i.e. $m!$), the first step is to generate a representative sample set of domain states. If the problem is to solve a specific instance of a domain the following simple algorithm already provides good results. As one is only interested in states reachable in the instance, it makes sense to start with a singleton set comprising of the initial state. At each iteration one randomly picks a state from this set, generates its successors and add them back to the set. The process ends after the set has either grown to a specified size or a fixed number of iterations have been executed, whichever happens first. The random selection is aimed at generating a good sample of valid states at different search depths.

Algorithm 10: PERM-SEARCH

Top-down, greedy search of sampleset for a permutation of bit positions that minimizes the entropy of subtree sizes at each tree level.

Input: sampleset a set of sampled states
Output: bitorder a permutation of the state encoding

subtrees $\leftarrow \{\text{sampleset}\}$;
bitorder $\leftarrow \langle\rangle$;

while *unassigned bit-positions* **do**
  foreach *unassigned bit-position* p **do**
    subtrees$^p \leftarrow \{\emptyset\}$;
    foreach $S \in$ subtrees **do**
      $S^p_{true} \leftarrow \{s \in S : s[p] = true\}$;
      $S^p_{false} \leftarrow S / S^p_{true}$;
      subtrees$^p \leftarrow$ subtrees$^p + S^p_{true} + S^p_{false}$;
    end
  end
  $p^* \leftarrow \underset{p}{\operatorname{argmin}}\{H(\text{subtrees}^p)\}$;
  bitorder $\leftarrow$ bitorder $\circ\, p^*$;
  subtrees $\leftarrow$ subtrees$^{p^*}$;
end

4.2.4 Analyzing the sample set

Then a suitable permutation or more precisely bijective mapping of bit-positions to tree levels can be deduced by greedily constructing a prefix tree over these sample states in top-down fashion. Algorithm 10 gives the pseudo code for perm-search. Each iteration begins with sets for each leaf node of the current tree, holding the subset with the prefix corresponding to the path from the root to the leaf node. The process starts with a single leaf-set comprising all states of the sample set, an empty bit-order and all bit-positions designated as candidates. During an iteration the algorithm examines each remaining unassigned candidate bit and computes the temporary new tree layer incidental to tis selection by partitioning each set according to the value of this bit in its states. To maximize average prefix lengths it selects the candidate with the least entropy in its leaf-sets as next in the bit-order. It terminates after m iterations, when all candidates have been assigned. Intuitively this process will move bits whose value is near-constant in the sample set to the most significant bit positions in the permuted string.

4.2.5 On Prefix Tree Encodings

While prefix trees can eliminate quite a bit of redundancy amongst set-members, standard in-memory representations often result in a size increase in comparison to a packed representation. The culprit are pointers, each of which require up to 8 bytes of storage on current machines (c.f. the 16 to 20 bytes required to store a BDD node). An alternative are pointerless structures such as the Ahnentafel[2] representation of binary heaps. A historical Ahnentafel established the generation order of individuals solely through their positions in the document. At the first position is the subject. The ordering rule is that for any individual at position i, the male ancestor can be found at position $2i$ and the female ancestor at position $2i + 1$ with the offspring to be found at position $\lfloor i/2 \rfloor$ (see Figure 4.8 for one of the first such documents).

In this way, a full binary tree is stored in an array through a bijection which maps its elements (individuals) to positions in the array in level-order (i.e. the order of their generation). Figure 4.9 gives an example. This technique is well suited for full binary trees (such as binary heaps), but not a good fit if the tree is sparse as most position are unused overhead. However the scheme can be adapted to the sparse (or general) case in a straightforward manner. The following encoding of general binary trees was supposedly first introduced by Knuth [Knu73]. Given some binary tree, each "missing" child is replaced with special terminating nodes. Then the tree is converted into a bit string by traversing it in left-to-right level-order. For each regular node a 1 bit is appended to the string and for each terminating node a 0 Bit is appended. An

[2]German for ancestor table.

FIGURE 4.8 The first *Ahnentafel* published by Michael Eytzinger in *Thesaurus principum hac aetate in Europa viventium* Cologne: 1590, pp. 146-147, showing the ancestry of Henry III of France. Note that for older generations on the right, the tree structure collapses into strings, in which the relative positions of the entities defines their relationships.

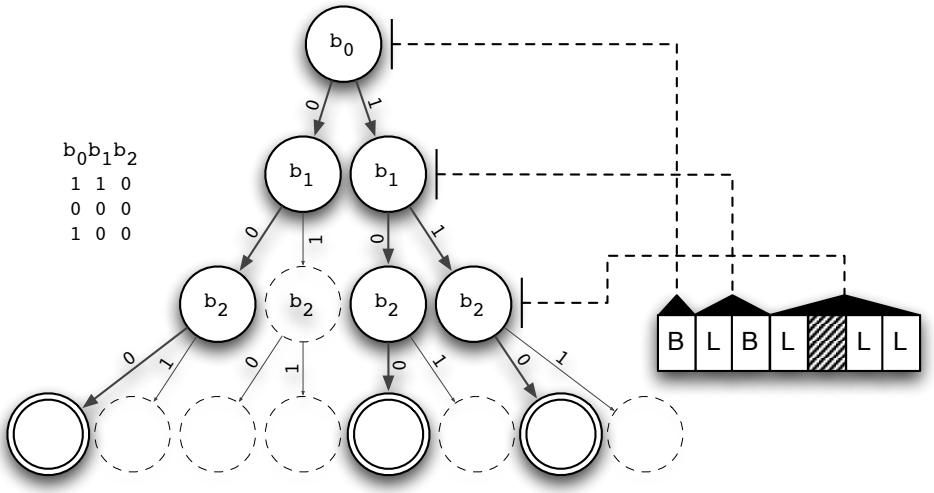


FIGURE 4.9 Binary prefix tree and corresponding *Ahnentafel* representation of a set of three states. Table entries discern between the presence of the left, right and both children at a node. Hatched entries represent unused overhead.

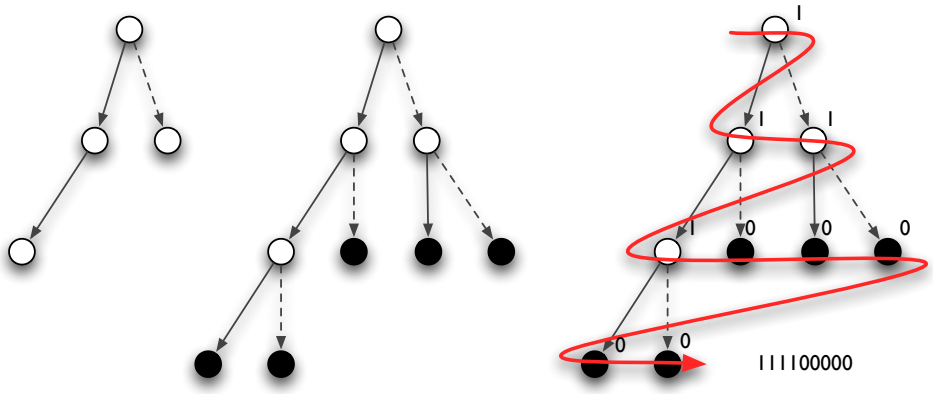


FIGURE 4.10 A sparse binary tree (left). The tree with added terminating nodes (middle). Binary code for the left-to-right level-order traversal (right).

example is given in figure 4.10. For an n node tree, this results in an $2n + 1$ length bit-string. The idea of an induction proof is as follows. A single node tree comprises of the node and two terminating nodes or a 3 bit string. If any $n - 1$ node tree can be encoded in $2n - 1$ bits, any n node tree can be encoded in $2n + 1$ bits by beginning with an $n - 1$ tree that differs in only one position, replacing the terminating node with the differing node and adding two new terminating nodes. The resulting bit-string hence grows by 2 bits.

As this is a level-order encoding, the following holds. Say the bit of a node is at some position p in the string , then i (i.e., its order or *rank* in the level-order traversal) can be determined by counting the number of set bits in the string from its beginning up to p. The children of this node can be found at positions $2i$ and $2i + 1$ in the encoding .

4.2.5.1 Binary Tree Encoding from the Perspective of Information Theory

Now a short information-theoretical digression on binary trees to show that such an encoding of the tree structure is close to optimal. The number of different binary trees with regards to their order (i.e. number of nodes) can be derived as follows [dS58]. There is only one empty and one single node tree configuration. Higher order trees always comprise of a root node and $n - 1$ nodes amongst its two subtrees. Thus if the first subtree has $i \in [0; n - 1]$ nodes, the second must be of order $n - 1 - i$. From these observations a recursive generating function (see 4.1) can be derived straightforwardly - the corresponding series is known as the *Catalan numbers* (note that the Catalan series plays a far more central role in combinatorics than this short section might suggest; for a more in depth treatment see [S+99]).

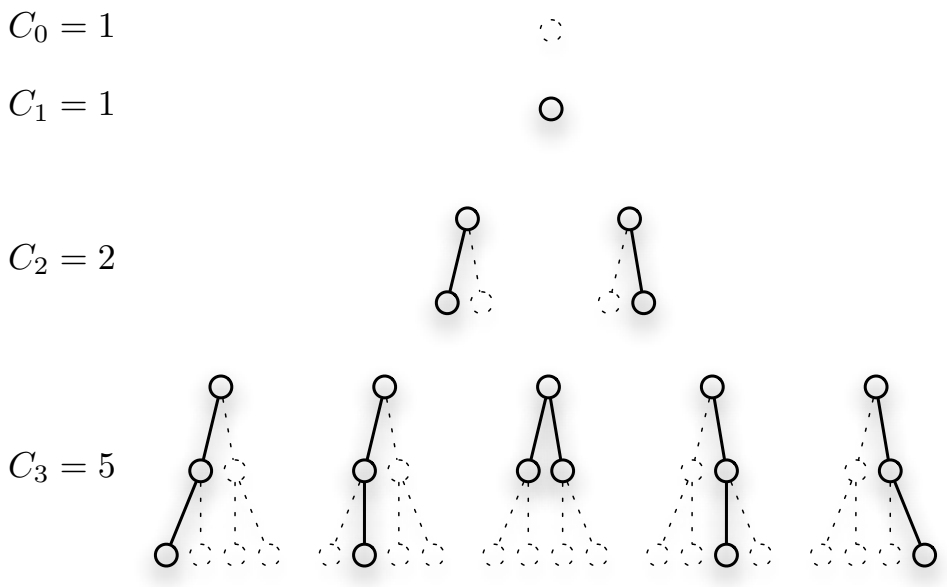


FIGURE 4.11 First four numbers of the Catalan Series and corresponding binary trees.

$$C_n = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{n-1} C_i C_{n-1-i} & \text{else} \end{cases} \tag{4.1}$$

Figure 4.11 gives C_0 to C_3 with the different specific binary trees of the corresponding degree. By considering some enumerative function that bijectively maps binary trees of order n to the co-domain $0 \ldots C_n - 1$, computing the base 2 logarithm of C_n gives the information theoretic minimum number of bits required to represent a general binary tree of n nodes in the "worst case" (i.e. by assuming all trees have an equal likeliness to occur). Stirling's approximation gives $\log_2 C_n$ as $2n + o(n)$ [3]. It is worth pointing out that general k-ary trees (where a node can have up to k children) can be bijectively mapped to binary trees of the same degree (and number of edges) using techniques such as *left-child right-sibling* (LCRS). Hence these deliberations extend to general trees of n nodes.

[3] note the *little-o* notation, i.e. $\lim_{n\to\infty} \frac{o(n)}{n} = 0$

4.2.5.2 Succinct Data Structures

Jacobson has introduced the term *succinct data structure* for any data type representation that uses space "close" to the information theoretic lower bound while allowing efficient in-place search and navigation [Jac88]. There are quite a few encodings that achieve $2n$ bits per node on ordered n-ary trees (where there is some total order defined for the children of a node). During encoding, the tree is usually traversed in *level-* or *pre-order* (breadth-first or depth-first order in search terminology). Nodes are commonly stored either as sequentially concatenated bit-strings of i $true$ bits terminated by a $false$ bit for a node with i children or nested pairs of parenthesis. Two well known succinct data structures based on such encodings are the Level-Ordered Unary Degree Sequence (LOUDS see [Jac88]) using level-ordered, concatenated node records and Balanced Parenthesis (BP, see [MR02]) using nested pairs of open and closed parenthesis in pre-order. Figure 4.12 gives the corresponding codes of both approaches for a small example tree. For LOUDS, each tree node (other than the root) accounts for a $true$ bit in its parent's record and a $false$ bit in its own record, for BP each node results in exactly 2 bits for the open and close parenthesis. Hence both can encode arbitrary ordered n-ary trees in $2n$ bits per node. As most succinct data structures, efficient navigation is implemented on top of two fundamental functions, *rank* and *select*. *Rank* computes the number of occurrences of a symbol in a string up to a given index and its inverse *select*, computes the index of the n-th occurrence of a symbol. Constant-time (and hence efficient) implementations for both functions are possible using directories of less than n bits (see [Eli74]), hence LOUDS and BP qualify as succinct structures. Such structures are used as static maps or dictionaries over semi-structured data such as large, static XML documents[DRR06].

4.2.5.3 Design Considerations

Level-ordered representations have led a somewhat obscure existence, as pre-order encodings support a superset of navigation operations efficiently. Yet for descending a tree by label sequence (in other words, a set-member test and hence the most critical operation in the planning context), it is usually the most efficient organization due its cache-friendly locality of reference over the first tree levels. A good empirical evaluation of the efficiency of common navigation primitives on different succinct tree representations can be found in [ACNS10]. Another focus in the design of LOES was to avoid the variable length node records used by all common succinct encodings, as they necessitate an efficient *select* implementation to determine record boundaries for the required navigation primitives. The required directory to support this amounts to a noticeable increase in the size of such structures. In contrast to the

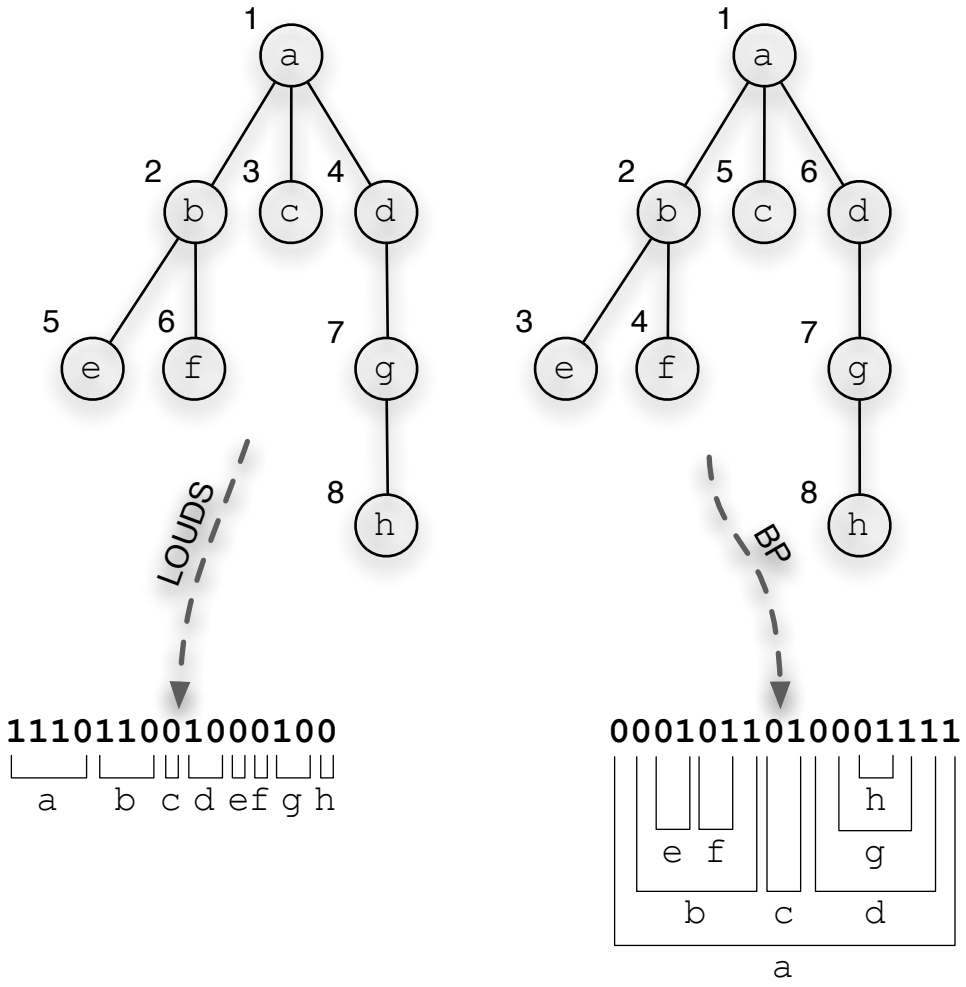


FIGURE 4.12 Level-order (equivalent to LOUDS without the abstract supernode as defined in [Jac89]) and balanced parenthesis encodings of an example tree. The numbers at each node give the respective encoding order.

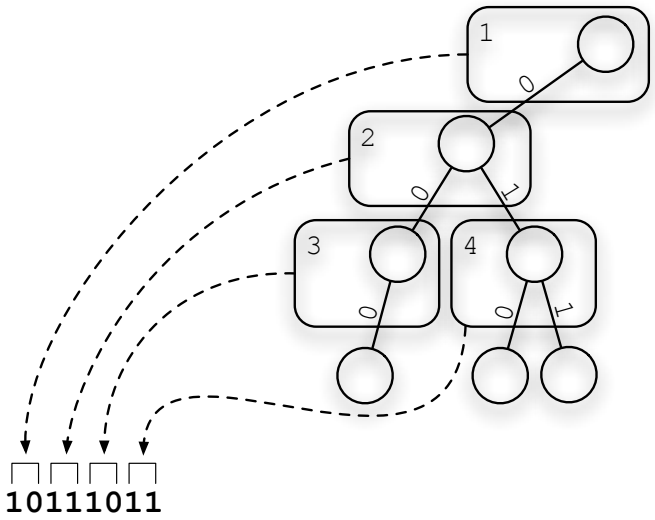


FIGURE 4.13 LOES code for the three state example set. The numbers at each node show the logical tree traversal during encoding.

ordinal LOUDS and BP encodings, which concern themselves solely with representing tree structure as a means to address node information in a separate array, LOES stores the entire (key)set information within the encoding and exploits the known size of the represented state encodings. Most importantly however it is designed in a way that it can accommodate updates to the key-set in amortized $O(\log(n))$ time.

4.2.6 The LOES Encoding

The logical tree encoding for LOES is straightforward. The prefix tree is traversed in level order and for each *inner node* a 2-bit record is appended to the LOES string. The first bit in the record denotes whether a $false$ edge originates from that node, the second bit analogously denotes the presence of a $true$ edge. This exploits the known tree depth, resulting in less than $2n$ bits in the code and comprises the entire information associated with the prefix tree, i.e. there is no need for an external label file to reconstruct the set from the LOES code. In fact, for the above type of prefix trees, the encoding resembles Knuth's when it is pruned of the first and last level of bits (as each node in the tree has exactly one incoming edge). Figure 4.13 shows how this allows to encode the example set in a single byte, little more than half the length of the more general encodings mentioned above.

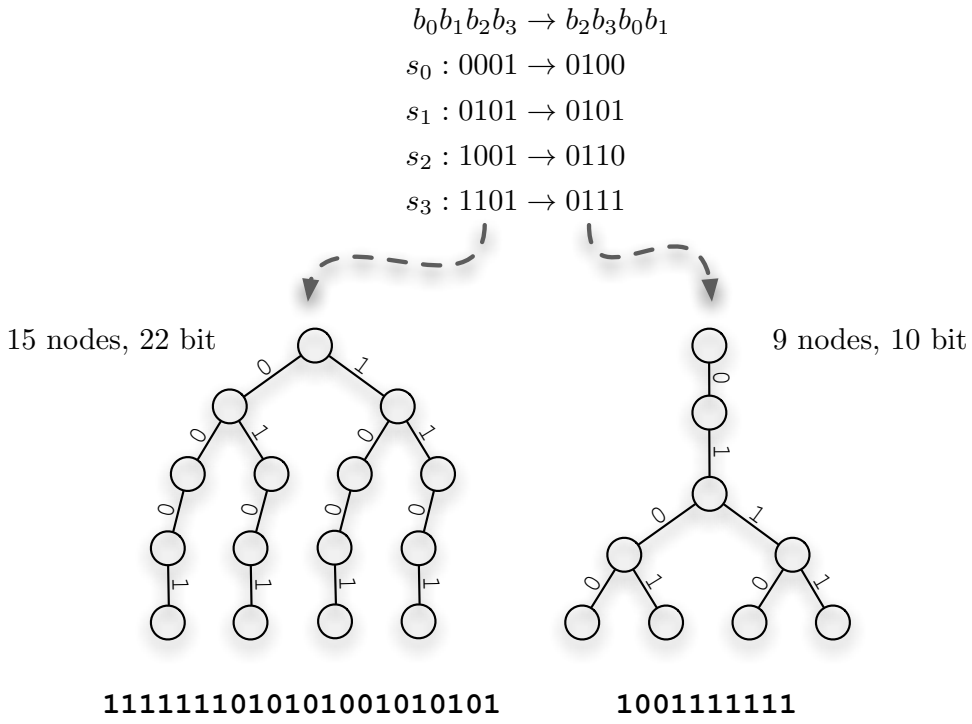


FIGURE 4.14 Encoding permutations for a set of 4 states resulting in worst and best-case prefix trees. The corresponding LOES is shown at the bottom.

4.2.7 Size Bounds of LOES Encodings

For a set of i unique states, the prefix tree is maximal, if the average common prefix length of the states is minimal. Intuitively this results in a structure that resembles a perfect binary tree up to depth $k = \lfloor \log_2 i \rfloor$ and degenerate trees from each of the nodes at depth k. Hence the set-tree will at worst encompass $i + i(m - k)$ nodes. For large sets of long states (i.e. $k = \log_2 i \ll m \ll i$) this is less than $(m+1)i$ nodes. As each node (with the exception of the tree root) has exactly one (incoming) edge and each record in LOES holds at least one edge, the code will at worst be about twice the length of the concatenation of packed states in the set. The complete formula is given as equation 4.2.

$$ub_{LOES} = \underbrace{2}_{\frac{bit}{node}} \Big(\underbrace{\underbrace{2i}_{0 \ldots \lfloor \log_2 i \rfloor} + \underbrace{i(m - \lfloor \log_2 i \rfloor)}_{\lfloor \log_2 i \rfloor + 1 \ldots m}}_{\text{inner nodes in the prefix tree}} - \underbrace{i}_{\text{leaf nodes}} \Big) \text{ bit} \leq 2i(m+1) \text{ bit} \qquad (4.2)$$

The best case results from the opposite situation, when the structure represents a degenerate tree up to depth $j = m - \lfloor \log_2 i \rfloor$, followed by a perfect binary tree on the lower levels. Such a tree comprises of $2i + (m - j)$ nodes, of with each record in the binary tree holds two edges.

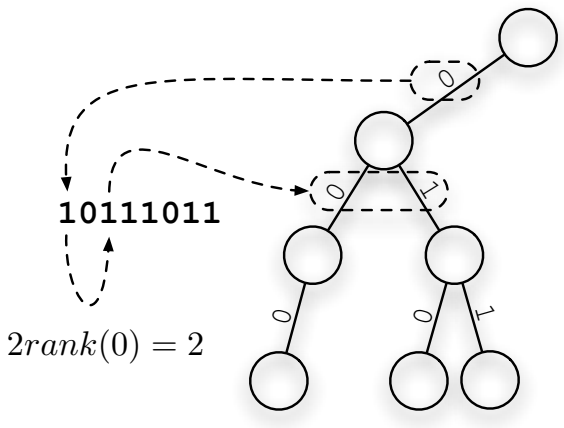


FIGURE 4.15 Tree navigation through the LOES using the *rank* function.

For large sets (i.e. $m \ll i$), $2i$ bits is hence a tight lower bound on the minimal length of the LOES code. Figure 4.14 gives an example and equation 4.3 the related formula.

$$lb_{LOES} = \underbrace{2}_{\frac{bit}{node}}(\underbrace{\underbrace{i\lfloor\log_2 i\rfloor}_{0\ldots\lfloor\log_2 i\rfloor} + \underbrace{2i}_{m-\lfloor\log_2 i\rfloor+1\ldots m} - \underbrace{i}_{\text{leaf nodes}}}_{\text{inner nodes in the prefix tree}})\text{bit} \geq 2i\text{bit} \tag{4.3}$$

4.2.8 Mapping Tree Navigation and Set Operations to LOES

For use in state-space search or planning a set/map data structure needs to support four operations in a time and space efficient way:

- Set-member queries.
- Bijective mapping of the set's i elements to integers $0 \ldots i-1$ to allow efficient association of ancillary data to states
- Iteration over the set elements.
- Insertion of new elements.

All of these operations require efficient navigation through the LOES string. The level-order encoding guarantees that for any edge in the sequence at some offset o, the record this edge points to points to can be found at offsets $2rank(o)$ to $2rank(o)+1$, where *rank* is a function that gives the number of set bits in the sequence up to (and including) offset i. This is, because each set bit (present edge) in the LOES code results in an edge-pair record for the target node

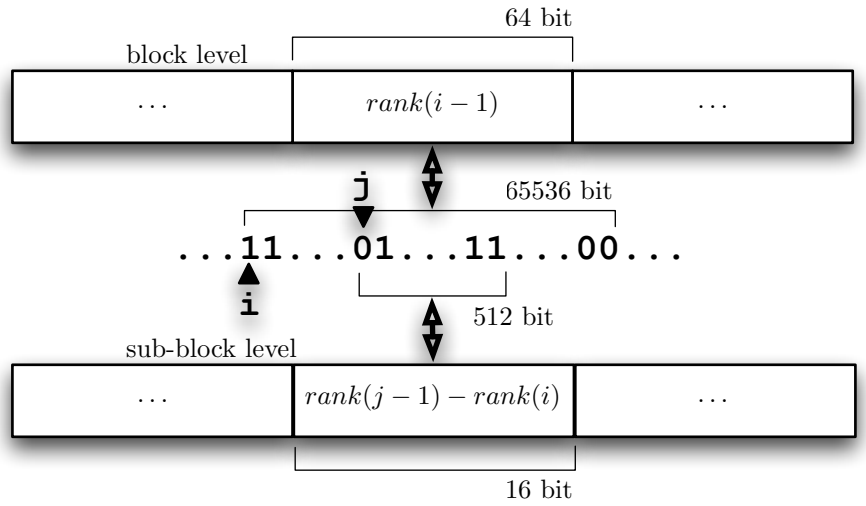


FIGURE 4.16 Structure of the two level *rank* dictionary.

on the next level (with the exception of the leaf level). As these records are stored in level order, all preceding (in the LOES) edges result in preceding child records. Hence the child record for some edge at offset o will be the $rank(o) + 1$-*th* record in the sequence (as the root node has no incoming edge). Transforming this to offsets with 2-bit records, $2rank(o)$ and $2rank(o) + 1$ then give the respective offsets of the edge's target node's $false$ and $true$ edges. Again, note the logical correspondence to the Ahnentafel encoding for binary heaps. Figure 4.15 shows this for the small example set.

4.2.8.1 Fast Rank Computation

The most basic implementation of *rank* is a linear scan of the LOES code. Navigating a tree top to bottom requires a rank computation at every level. As sets and hence the code can get very large, path computation with $O(nm)$ cost operators is not feasible in practice. An appropriate implementation of *rank* is backed by a two-level dictionary, which logically partitions the LOES into blocks of 2^{16} bits and sub-blocks of 512 bit. For each block, the index holds an 8-byte unsigned integer, denoting the number of set bits from the beginning of the sequence up to the beginning of the block. On the sub-block level, a 2-byte unsigned integer for each sub-block stores the number of set bits from the beginning of the enveloping block up to the beginning of the sub-block. Figure 4.16 gives a graphical representation of this structure. The

relative block overhead of this dictionary is hence

$$\underbrace{\frac{64}{2^{16}}}_{\text{block index}} + \underbrace{\frac{16}{2^{9}}}_{\text{sub-block index}} \approx 0.0323 \tag{4.4}$$

The dictionary can be constructed quickly with a linear scan through the LOES. I omit the pseudocode as it is extremely straightforward.

Algorithm 11: RANK

The *rank* function computes the number of set bits from the beginning of a sequence up to (and including) offset.

Input: offset an offset
Output: number of set bits in LOES$[0;$ offset$]$
Data: LOES an encoded state-set
Data: blocks block level of the dictionary
Data: subblocks sub-block level of the dictionary

```
rank ← blocks[⌊offset / 2^16⌋] ;                    /* blocks[offset ≫ 16] */
rank ← rank + subblocks[⌊offset / 2^9⌋] ;           /* blocks[offset ≫ 9] */
for i ← 2^3 ⌊offset / 2^9⌋ to ⌊offset / 2^6⌋ do
    w ← LOES[i] ;                                   /* next uint64 of the LOES */
    if i = ⌊offset / 2^6⌋ then                      /* prune extra bits */
        w ← w / 2^(63 − offset mod 64) ;            /* w ≫ (63 − (offset ∧_bw 0x3F)) */
    end
    rank ← rank + popcount(w);
end
return rank;
```

With this dictionary, the rank function comprises of straightforward look-ups of the block and sub-block indices and set-bit counting within the sub-block. As the operation is critical for the performance of all set primitives, algorithm 11 gives an exhaustive description for an implementation on a 64-bit machine (assuming bit addressing from *most* to *least* significant bit within a word). Note that necessary computations map to a few memory-lookups, single-cycle shifts and bitwise-conjunctions as indicated by the comments.

The popcount function computes the number of set bits in a machine word. This is implemented in hardware for many current consumer level CPUs [4]. For the evaluated implementation, I used a simple software fallback given in listing 12. For formatting reasons, the

[4] the instruction is optional on the common INTEL® IA64 and AMD® x86-64 ISAs, support is indicated by the `CPUID.01H:ECX.POPCNT[Bit 23]` flag.

Algorithm 12: POPCOUNT

Software implementation of popcount on 32-bit words.

Input: value a 32-bit integer
Output: count number of set bits in value

```
// 16 pair counts
1 value ← value − ((value ≫ 1) ∧bw 0x55555555);
// 8 quad counts
2 value ← (value ∧bw 0x33333333) + ((value ≫ 2) ∧bw 0x33333333);
// 4 octet counts, accumulate to highest byte and normalize
3 count ← ((value + (value ≫ 4) ∧bw 0x0F0F0F0F)0x01010101) ≫ 24;
```

algorithm is given for 32-Bit integers, but straightforwardly extends to larger types. In the first line, all 16 2-bit pairs are counted. In the second line these are combined to 8 4-bit counts and then again to 4 8-bit counts in the final line, after which all bytes are accumulated to the most significant byte which is then normalized through a shift. More efficient implementations exist for larger chunks (see [EQ01]). Implementing rank like this results in an $O(1)$ operator with a very small constant and makes it an efficient building block for the required set operations.

4.2.8.2 The Path-Offset Function

The first such set operation is the path-offset function (Algorithm 13). It navigates through the LOES according to the label-sequence interpretation of a state. Logically it begins at the root of the prefix tree represented by the LOES. If the state maps to a valid path from the tree root to some leaf, path-offset returns the offset of the bit corresponding to the last edge of the path. Else it evaluates to $\perp$. An example is given in figure 4.17

4.2.8.3 Set-Member Testing

With the path-offset function in place, member tests are straightforward (see Listing 14). As set contains a state, if and only if its label interpretation corresponds to a valid path through the prefix tree, one simply needs to check whether path-offset returns a valid offset.

4.2.8.4 Set-Member Indexing

The member index function (Algorithm 15) maps states to values $\{\perp, 0, \ldots, n-1\}$. It is bijective for all member states of the set and hence allows associating ancillary data for each member through offset addressing and hence without resorting to pointers. The idea is that

Algorithm 13: PATH-OFFSET

given an encoded state, path-offset navigates the LOES according to the states path interpretation

Input: state a bitsequence of length m
Input: LOES an encoded state-set
Output: offset an offset into LOES or $\perp$

```
offset ← 0;
for depth ← 0 to m − 1 do
    if state[depth] then
        offset ← offset + 1;
    end
    if LOES[offset] then
        if depth = m − 1 then
            return offset;
        end
        else
            offset ← 2rank_LOES(offset);
        end
    end
    else
        return ⊥;
    end
end
```

Algorithm 14: MEMBER-TEST

Member-test computes whether state is in the set encoded by LOES.

Input: state a bitsequence of length m
Input: LOES an encoded state-set
Output: $true$ iff state is represented in LOES

return (`path-offset (`LOES, state `)` $\neq \perp$);

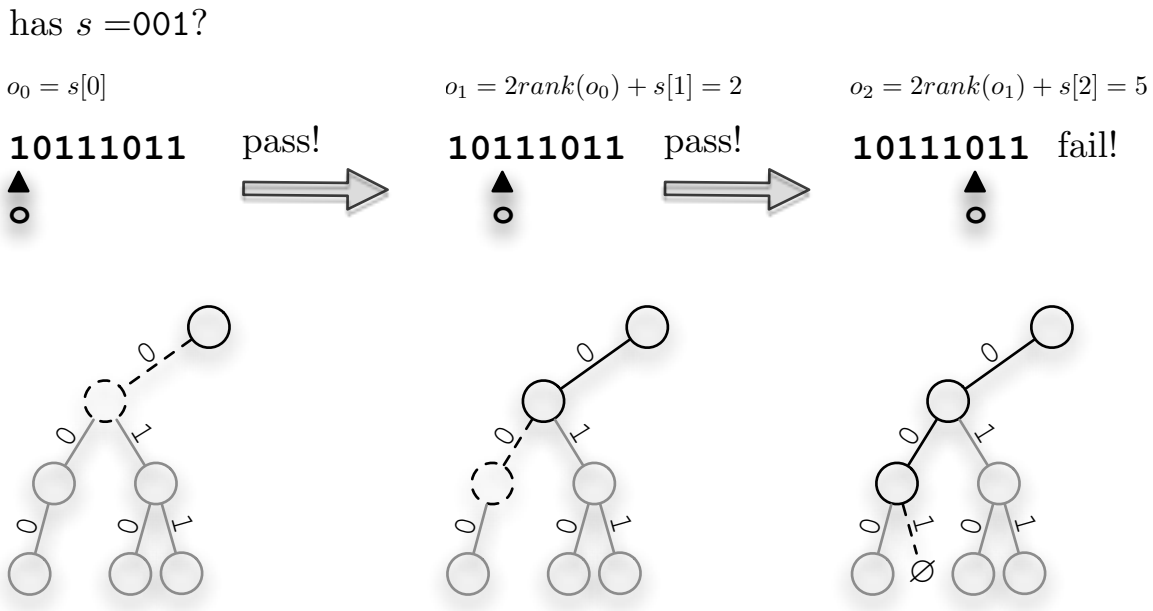


FIGURE 4.17 Path-offset computation for an encoded state $s = 001$ over a LOES code corresponding to the example set. At each level, the offset of the associated edge-presence bit is computed and the LOES tested at the offset. If the corresponding bit is set, the process continues at the next level (or returns the offset at the last level), else $\perp$ is returned.

Algorithm 15: MEMBER-INDEX

Member-index maps state to $\{\perp, 0, \ldots, n-1\}$ such that for all states in LOES, the mapping is bijective.

Input: state a bitsequence of length m
Data: LOES an encoded state-set
Data: levelOffsets array of offsets denoting the positions of the last set bits at each level

$o \leftarrow$ `path-offset(`state`);`
if $o = \perp$ **then**
  return $\perp$;
end
$a \leftarrow \texttt{rank}_{\text{LOES}}(o)$;
$b \leftarrow \texttt{rank}_{\text{LOES}}(\text{levelOffsets}[m-1]-1)$;
return $a - b - 1$;

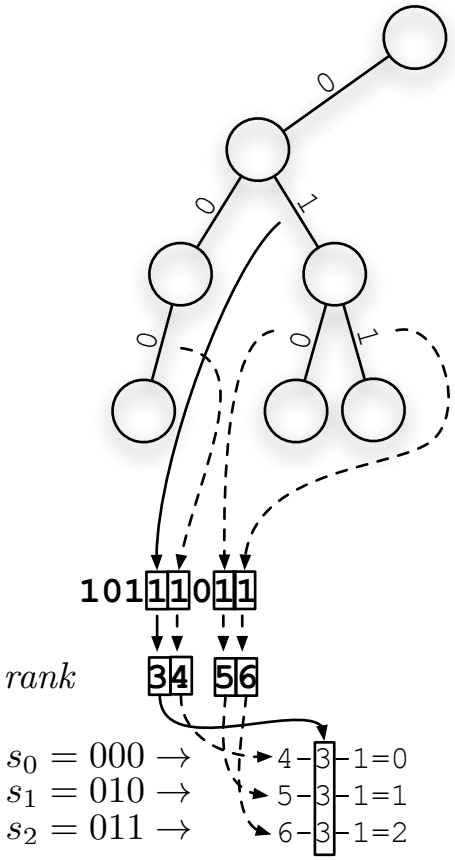


FIGURE 4.18 Index mappings for all states in the example set. One subtracts rank+1 (of the offset) of the last edge in the last-but-one level from the rank of the path-offset of an element to compute its index.

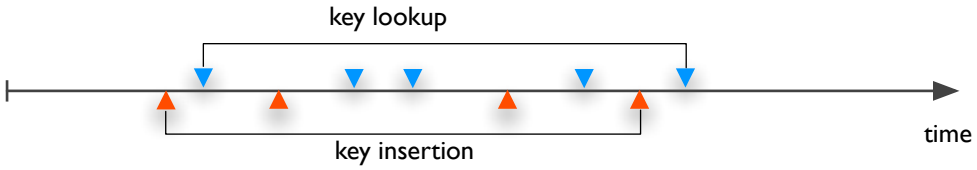


FIGURE 4.19 The lifecycle of a memoization component in dynamic programming comprises of key lookups to retrieve or update the associated data, as well as insertions of new key-data pairs.

the path interpretation of each state in the set ends in a unique (to this state) edge at the leaf-level of the tree. The ranks of these bits in the LOES form a consecutive range, One can use path-offset to compute the associated offset of that bit for a member state. Computing the rank of that offset, gives the corresponding unique integer for that state. These values can be normalized to the desired $[0; n)$ interval by subtracting the rank of the last offset of the last but one layer plus one (i.e. the rank of the first set bit in the last layer). Figure 4.18 shows how this uniquely maps constituents of the example set to the range $[0; 2]$.

4.2.9 Building a Dynamic Data-Structure for Dynamic Programming based on LOES

The LOES code as described so far is a domain-specific adaptation of familiar level-order tree encodings. As other approaches, it is static as additions of an element necessitate changes to the bit-string that are not locally confined and the cost of such naive insertion is hence $O(n)$ making its direct use unpractical for scenarios where the tree structure changes regularly such as dynamic programming where new keys have to be inserted regularly. This is particularly true in the large problems LOES is aimed at where the key set can comprise of hundreds of millions of states. In the following, I will show that for the purposes of dynamic programming, a memoization component based on LOES can be constructed that guarantees worst-case $O(\log(n))$ lookup and *insertion* costs. For brevity and clarity (and following the usual conventions) I refer to any operation that only depends on the complexity of keys (i.e. the length of their bit-sequences) as constant time.

First, figure 4.19 gives an example at how a memorization component is employed in dynamic programming. Over its lifecycle, data is looked up for keys, new data is associated with keys and finally new key-data pairs are inserted. Newly discovered subproblems (i.e. keys) are generally never discarded in DP, making the memorization component *grow monotonically* over time. Note that the first two operations do not change the key set in any way and are

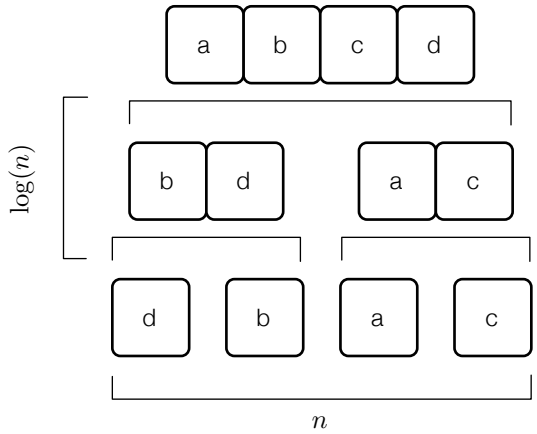


FIGURE 4.20 In a merge-sort, beginning from n single element sequences (which are trivially sorted), sorted sequences of size k are iteratively combined to sorted sequence of size $2k$ until the final sorted n element sequence is constructed.

covered by the above query algorithms in $O(1)$ time (i.e. their complexity is independent of the number of keys in the set).

To see how this lifecycle can be exploited consider the standard merge-sort algorithm [Knu73]. In a merge sort, a sequence of n elements is first split into n singleton sequences that are sorted by default. Then in each iteration, pairs of sorted sequences of size k are converted into a sorted sequence of $2k$ elements. Figure 4.20 shows this graphically. Such a transformation can be computed in linear time in the length of both sequences. As there are $\log(n)$ iterations and each iteration comprises of sequences of a combined length n, merge-sort is an $O(n \log(n))$ algorithm.

For now I want to assume, that two LOES sequences of n and m keys over compatible key-spaces can be "merged" into a new LOES sequence comprising the union of both keys in an $O(n + m)$ operation. Now consider an memorization component that comprises of an array of LOES sequences where the position of a LOES corresponds to the number of merges the set has participated in. When a new key is added to this structure, it is first converted into a LOES code with merge count 0. If the corresponding position in the array is free, the new LOES is simply inserted in this position. Otherwise, it is merged with the prior occupant, its merge count incremented and the insertion-merge process continues.

Note that over the lifecycle of the component, this resembles a merge sort and inherits its complexity guarantees under the assumption of linear cost set merges and constant cost singleton construction. In other words this scheme *guarantees* a worst case insertion complexity of $O(n \log(n))$ after n insertions and hence an amortized $O(\log(n))$ complexity per insertion.

The price is that lookups and data-changes now have to be executed against multiple LOES sets, but this scheme guarantees that such a composite set of n nodes can comprise of *no more than* $\log(n)$ *LOES codes* at any time. As the lookup operations only depend on key complexity, these operations are $O(\log(n))$ in the worst case. In other words, the amortized complexities of queries and insertions are the same as for pointer based tree-structures such as AVL- or Red-Black trees.

Figure 4.21 shows the transformations and their corresponding merge-sort analogues for the example lifecycle given in figure 4.19. Note that the *amortized* insertion cost and query cost per element is upper bounded by $O(\log n)$ at any time in the lifecycle. A singleton LOES set can be trivially constructed from an m bit key, by concatenating m 10 and 01 records depending on the bit configuration. For a fixed key-length, this operation is trivially constant time. What remains is to devise a way to merge two LOES encoded sets in linear time. To elaborate on this topic, I first want to show how the keys encoded in a LOES can be retrieved in lexicographic order in linear time.

4.2.9.1 Lexicographic Set-Member Enumeration

Algorithm 16: IT-ADVANCE

Given an iteration state offsets corresponding to a LOES element, It-advance modifies this state such that it corresponds to the next element in the LOES.

Data: LOES an encoded state-set
Data: offsets an array of LOES offsets (Iteration state)

level $\leftarrow m - 1$;
continue $\leftarrow$ true;

for continue & level ≥ 0 **do**
  $\text{rec}_{id} \leftarrow \lfloor \frac{\text{offsets[level]}}{2} \rfloor$;
  repeat
    offsets[level] $\leftarrow$ offsets[level] $+ 1$;
  until LOES[offsets[level]] $= true$;
  continue $\leftarrow \text{rec}_{id} \neq \lfloor \frac{\text{offsets[level]}}{2} \rfloor$;
  level $\leftarrow$ level $- 1$;
end

This iteration works by logically iterating over the set bits in the LOES subranges corresponding to the levels of the tree in parallel. Iteration state is represented by an array of offsets into the LOES, one for each level. This state is initialized with the offsets of the first set bit in each level, that is the path interpretation of the first set element. These offsets can be computed

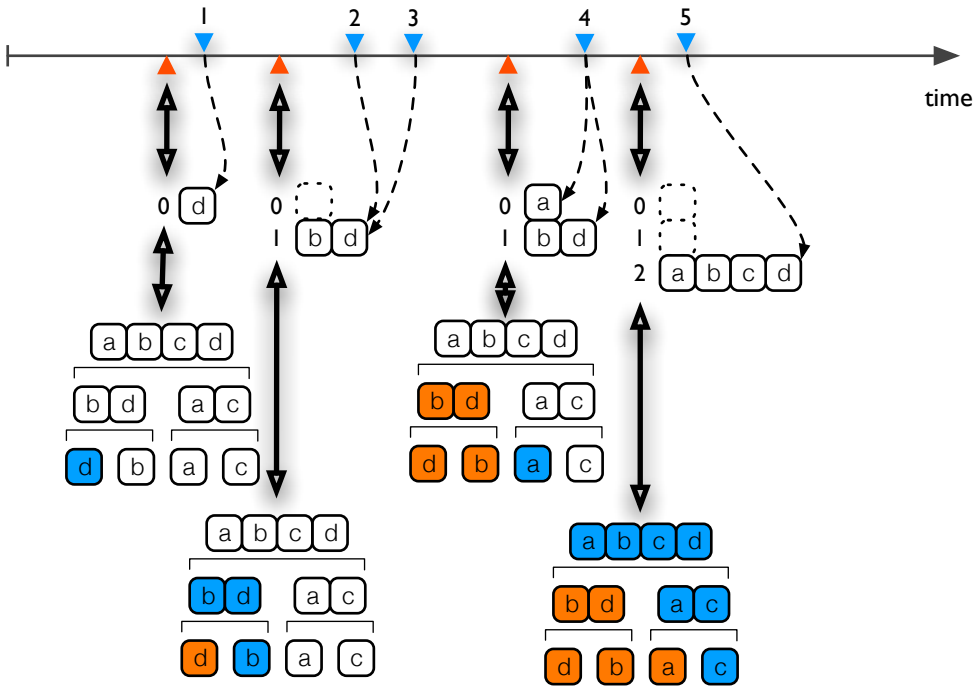


FIGURE 4.21 Given an example history of updates and queries, the above diagram shows the LOES file configuration after each update and the analogous merge-sort operations corresponding to the update in blue. Operations corresponding to previous updates are shown in orange. The first key-insertion creates a singleton LOES code from key d. As the 0-merge position in the merge-array is empty the code is directly inserted. The second insertion starts out the same, but as the 0 position is now occupied by d, the d code is removed, merged with the b code and the resulting b, d code inserted at the 1-merge position. Insertion of a works analogously to insertion of d. Note that now multiple positions in the merge-array are occupied, hence query 4 has to be executed against both the a and b, d codes, but as each atomic query is $O(1)$ the $\lceil \log_2(n) \rceil = O(\log(n))$ worst-case complexity guarantee is upheld. The insertion of key c finally leads to two cascading merges. As 0 is occupied in the array, first the a, c merged code is created and as 1 is occupied, a second merge takes place and the resulting a, b, c, d code is stored at the (free) 2-merge position.

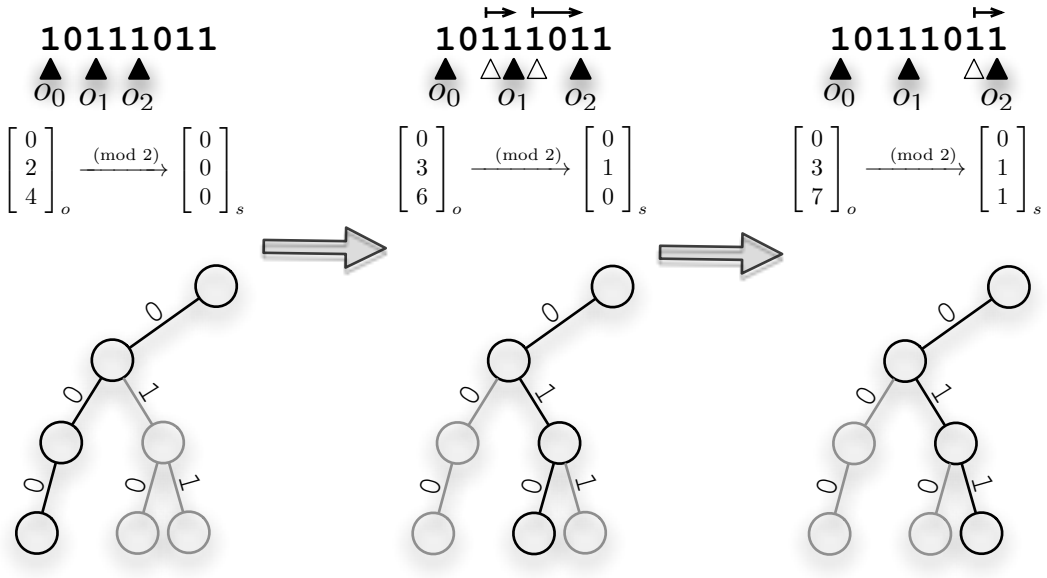


FIGURE 4.22 Iteration over the example set. The first row shows the LOES offsets (i.e. the iterator state) at each iteration stage. The second row shows how the associated element in the LOES can be reconstructed by field-wise $(\mathrm{mod}\ 2)$ application to these offsets. A graphical representation of the prefix tree with the respective element highlighted is given in the last row.

by single walk from the root to the first leaf (i.e. following the first set bit at each level) in O(1) (as individual steps take constant time and the tree depth is only dependent on key complexity). The iteration finishes when the leaf-level offset is advanced past the last set bit in the LOES. Algorithm 16 gives the pseudocode for advancing the iteration state from one element to the next element in the set. Beginning at the leaf-level, the respective offset is advanced to the next set bit. If the offset passes a record boundary (every 2-bits) the process continues at the next higher level. As at least one bit is set per record, the operation comprises no more than $2m$ offset increments. Figure 4.22 shows the different iteration states for our running example.

Algorithm 17 shows how to reconstruct the represented element given the offsets of the edges corresponding to its path interpretation at each level. It exploits the invariant that even-offsets into the LOES always correspond to $false$ labels and odd-offsets to $true$ labels. Hence, a simple modulo 2 of the offset will produce the corresponding bit assignment. Note that the iteration scheme guarantees to return keys in ascending lexicographical order. Over the course of the iteration each offset variable traverses over its corresponding level and looks up the corresponding bit values, that is all positions in the string will be referenced and accessed exactly once by some offset. As the worst-case length of a LOES code is linearly bound by the

Algorithm 17: IT-EXTRACT

Given an iteration state offsets corresponding to a LOES element, It-extract reconstructs the original bit-string of the element.

Output: state a bitstring of length m
Data: offsets an array of LOES offsets (Iteration state)

```
for i ← 0 to m − 1 do
    if offsets[i] mod 2 then
        state [i] ← 1;
    end
    else
        state [i] ← 0;
    end
end
```

number of its represented keys, the complexity of lexicographic enumeration is $O(n)$. Given two lexicographically ordered sequences of elements of sizes i and j, the merged lexicographical sequence of length $i + j$ can be trivially constructed in linear time, by comparing the two next elements of both sequences, choosing the precedent according to lexicographic order and advancing the iteration of the corresponding sequence. This scheme results in a total of $i + j$ iteration steps, each of which comprises of a constant cost key comparison (in regards to the number of elements in the sequence) and a constant cost iterator advancement. Hence the "merged" lexicographic sequence of keys of two LOES sets of i and j keys can be generated $O(i + j)$ time.

4.2.10 Construction of a LOES code from a Lexicographically-ordered Key-Sequence

What remains to be shown is that such a sequence of *lexicographically ordered* states can be transformed into a LOES code in linear time. The idea is as follows. Construction begins with empty bit-sequences for each layer of the tree. Algorithm 18 shows how these sequences are manipulated when adding a new state. If the set is empty, corresponding records are merely appended on all levels. Later insertions start with determining the position or depth d of the first differing bit between s', the lexicographically largest element in the LOES (i.e. the last insertion) and s, the element to be inserted. Then, the last bit of sequence d is set to $true$. Finally corresponding (to s) records are appended to all bit-srings $> d$. Note that duplicates (i.e. $s = s'$) are automatically ignored by this process. After the last state has been added, all

Algorithm 18: ADD-STATE

Given a LOES code partitioned by tree level and a state *lexicographically* greater than the states represented in the LOES, add-state manipulates the code partitions, such that their concatenation contains the state (in addition to all prior constituents).

Input: s a bitsequence of length m
Data: treelevels an array of bitsequences
Data: s' a bitsequence of length m or $\perp$

```
if s' = ⊥ then
    depth ← −1;
end
if s' = s then
    return;
end
else
    depth ← i : ∀j < i, s[j] = s'[j] ∧ s[i] ≠ s'[i];
    treelevels[depth].lastBit ← true;
end
for i ← depth + 1 to m − 1 do
    if s [i] then
        treelevels[i] ← treelevels[i] ∘ ⟨01⟩;
    end
    else
        treelevels[i] ← treelevels[i] ∘ ⟨10⟩;
    end
end
s' ← s;
```

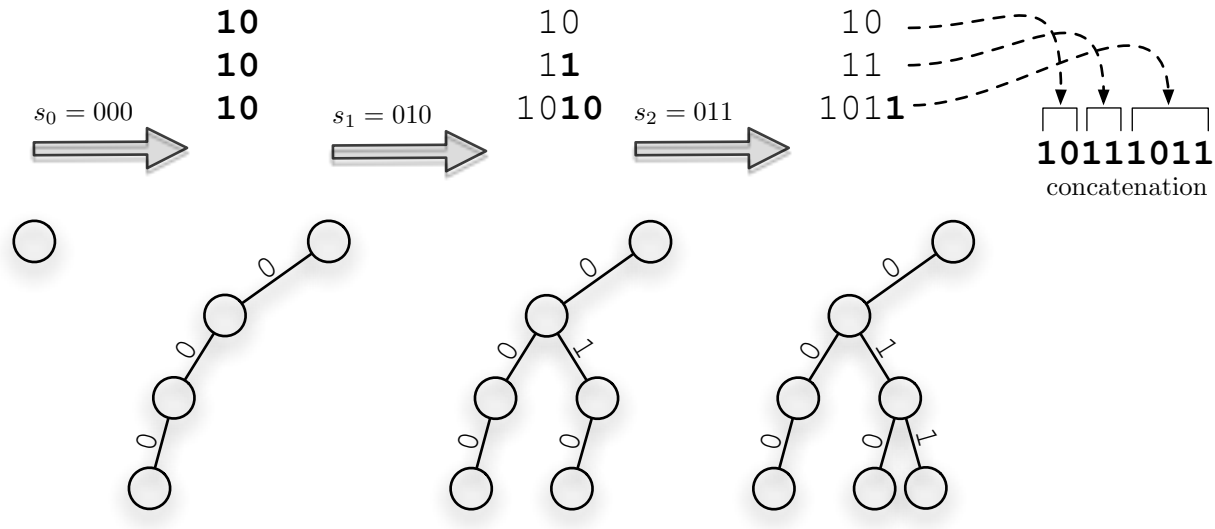


FIGURE 4.23 Construction of the set by adding the example states in lexicographic order with algorithm **??**. After all states are added, the edge sequences are concatenated in level-order to form the LOES.

sequences are concatenated in level-order to form the LOES. Figure 4.23 shows this process for our running example. Note that for each insertion consists of at most m constant time string operations, where m is the complexity of a key (i.e. the number of bits in its encoding). The computational complexity of each insertion step is hence independent of the number of keys in the set. For a set of n keys there are n insertion steps plus the one final concatenation of bit strings. The total size of these bit strings is a linear function of n. Hence the total computational complexity of this operation is $O(n)$.

4.2.11 Virtual In-Place Merging

With these operations in place, LOES can serve as a *black-box map implementation for dynamic programming with amortized* $O(\log(n))$ *test, update and insertion operations*. But a potential problem remains. In the worst case (consider the case when the merged-indexed LOES array is completely filled), a single insertion can lead to temporary doubling in required memory as the two largest order sets are merged. What is needed is a way to free memory occupied by the two source sets of the merge as it is allocated to the growing target set. This can be done by exploiting the fact that memory accesses are entirely predictable during an iteration. Per level, bit-string positions are accessed in linear sweeps, hence memory associated to previously accessed positions can be safely freed. For obvious reasons, memory handling

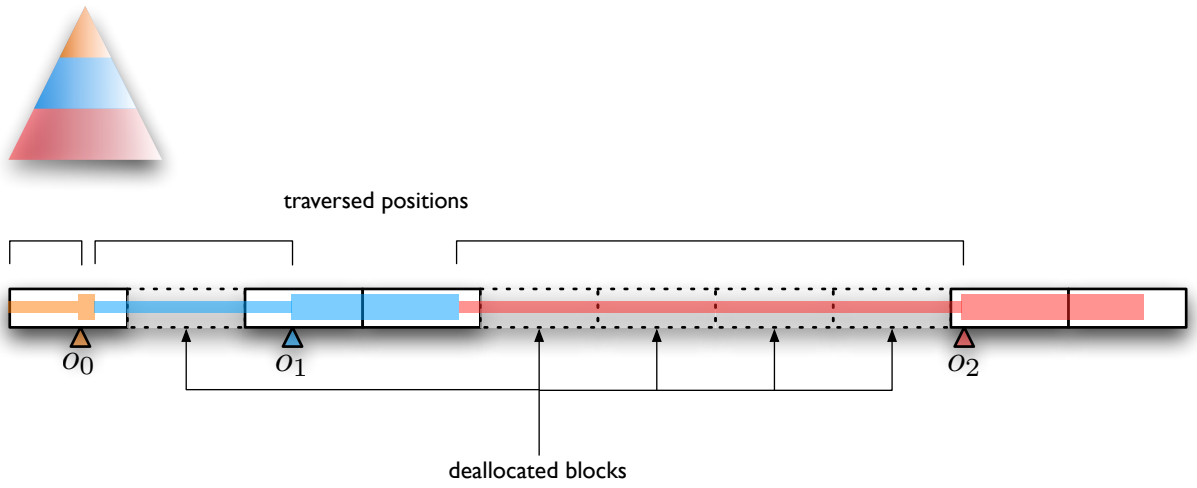


FIGURE 4.24 As a LOES code is accessed in multiple disjunct linear scans during iteration, memory blocks can be freed as soon as all containing bits have been accessed once. The above figure exemplifies this. In this situation, dotted blocks can be freed as they have been completely traversed.

on the bit level is impractical, so assume all bit sequences are logically backed by blocks that are managed by some segregated store. Based on this one can implement a destructive iteration, that produces all keys in lexicographic order as specified above, but simultaneously relinquishes ownership of these blocks as soon as it has been completely covered by all corresponding linear sweeps. While this scheme differs from the usual interpretation of "in-place" operations on first sight, it is related in spirit as the net memory overhead during the transformation is a small number of blocks, only dependent on key complexity (i.e. the number of levels in the tree). As the level i string of a merged LOES sequence can be at most the sum of the length of the level i strings in its two constituent sequences, the worst case overhead is $4m$ blocks, where m is the key size in bits. Figure 4.24 gives an example.

4.2.12 Practical Optimizations

An easy practical optimization to trade space-efficiency for faster runtime is to represent the lower order merged sets in a compatible data structure with a lower constant overhead and higher space cost. Once a set size limit for such a buffer is reached, it is transformed into a LOES code and from there on treated in the fashion above.

LOES integrates straightforwardly into unit-cost search algorithms. As discussed in the beginning of this chapter, *Open* comprises of relatively few, individually large sets of equal f-estimate candidates. For brevity, I here only sketch the integration of LOES into BFS-DD

Algorithm 19: BFS-DD-LOES

Breadth-first search with duplicate detection and bulk operations;

Output: an optimal i-plan or $\perp$

$Open_0 \leftarrow \{i\}$;
$Closed \leftarrow \emptyset$;
$Dict[i] \leftarrow []$;
$d \leftarrow 0$;

while $Open \neq \emptyset$ **do**
  foreach $s \in Open_d (\equiv Frontier)$ **do**
    foreach $o \in O$ *s.th.* $\delta(s, o) \neq s$ **do**
      $s' \leftarrow \delta(s, o)$;
      if $s' \in G$ **then**
        return $Dict[s] \circ o$;
      end
      if $s' \notin Open \cup Closed$ **then**
        $Dict[s'] \leftarrow Dict[s] \circ o$;
        $Open_{d+1} \leftarrow Open_{d+1} \cup \{s'\}$;
      end
    end
  end
  $Closed \leftarrow Closed \cup Open_d$;
  $Open \leftarrow Open \setminus Open_d$;
  $d \leftarrow d + 1$;
end
return $\perp$;

(c.f. section 2.3.2 and particularly algorithm 3). Heuristic best-first searches work analogously. In BFS-DD $Open$ comprises (at most) of two layers, the frontier and unique states generated during expansion of the frontier. The only optimization to the algorithm (see Listing 19) is that states are not individually moved from $Open$ to $Close$ during layer expansion which would necessitate the support of efficient deletion, but are moved in bulk after the last state has been expanded. While I omit details in the listing, LOES significantly simplifies the dictionary design. It comprises separate stores for $Open$ and $Close$, each of which holds a sequence of records of ancillary data (preferably in a packed representation) for the corresponding states. Organization of these sequences is mandated through the member-index function.

4.2.13 Empirical Comparison of LOES and BDD in BFS-DD

The following empirical evaluation concentrates on (peak-)memory requirements during blind searches in a range of IPC[5] domains. To this end, I implemented a breath-first search environment for comparative evaluation of LOES and BDD based representations. The BDD version is based on the BuDDy package [LNAH+01], a high-performance implementation from the model-checking community. Both representations provide an identical interface and are treated as black-boxes by the BFS-DD like algorithm (i.e. the LOES version used the scheme above with virtual in-place merges and a small insertion buffer backed by an STL map, iteration through the BDD was implemented by computing its satisfying set). In contrast to BFS-DD, I did not use a per-state dictionary in this evaluation as it would have put the BDD at a severe (spatial efficiency) disadvantage. Instead I organized $Close$ as a set of "former" frontiers. Note that the solution can be reconstructed once a goal is found by beginning from a singleton set comprising the goal, repeatedly computing the predecessor set and its intersection with the corresponding frontier in $Close$ until the initial state is discovered. While a somewhat clumsy technique it allows for a good black-box comparison of the two representations within identical test implementations. Note that this results in multiple membership tests for each $Close$ lookup (i.e. one for each prior layer), a significant overhead that only results from making the testbed BDD-"friendly". Hence the upside and motivation for this design is that resulting algorithm provides realistic, identical loads with equivalent operations to both representations. A direct consequence however is that the time comparison to FD is strongly biased towards the latter (i.e. each $Close$ lookup in layer n of FD corresponds to $n-1$ lookups for the LOES and BDD implementations). The tests were run on (identical) workstations equipped with Intel 2.26 GHz Xeon processors and 8GB of RAM.

[5] International Planning Competition

Instance	m_{loes}	m_{fd}	$\frac{m_{loes}}{m_{fd}}$
Blocks-9-0	46.8	1460.3	0.03
Depots-3	17.8	737.2	0.02
Driverlog-7	37.9	3686.1	0.01
Freecell-4	53.7	1092.9	0.05
Gripper-7	13.7	1720.5	0.01
Microban-5	31.6	682.9	0.05
Satellite-4	18.6	101.8	0.18
Travel-6	22.1	225.1	0.10
Airport-9	11.4	171.9	0.07

TABLE 4.1 Peak allocated process memory for LOES and FD on the largest instances FD could solve in MB.

4.2.13.1 FD's Time-Space Tradeoff

To provide some context, I ran Fast Downward (FD) in its blind-heuristic mode ($h = 0$) over the same instances. FD uses STL containers for all representation tasks, which results in very fast but large state, set and queue representations. Table 4.1 compares the peak memory requirements of LOES and FD for the largest instances in each domain FD could solve.

The algorithms used SAS+ representations generated by FD's preprocessor as input. For the BDD a variable-reordering using the package's recommended heuristic was initiated after each layer. As the order of expansions within a layer is undefined in breadth-first search and in practice generally depends on the inherent (i.e. most efficient) iteration order of the data structure, the total number of expansions differs for FD, BDD and *LOES*. A natural comparison point is after the last but one layer is fully expanded. For spatial reference, I also give the size of a corresponding, *idealized* (i.e. no padding or other overhead) concatenation of the (optimally) packed states in $Open$ and $Close$ (Packed). Note that, Packed also does not include overhead for buffers and other ancillary structures that the LOES and BDD include and does *not* represent FD's space requirements (c.f. table 4.1).

4.2.13.2 Results by Domain

Tables 4.2 to 4.16 give the results by domain. Individual states on average required between $6 - 58\%$ of the memory of the ideally packed representation on the larger instances of all

airport	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
P1	10	0.00	0.00	0.02	0	0
P2	15	0.00	0.00	0.02	0	0
P3	175	0.00	0.00	0.08	0	0.04
P4	22	0.00	0.00	0.02	0	0.03
P5	30	0.00	0.00	0.04	0	0.03
P6	765	0.01	0.01	0.61	0	0.8
P7	765	0.01	0.01	0.61	0	0.8
P8	27,458	0.57	0.26	9.82	1	43.92
P9	177,075	4.60	1.54	100.32	5	395.58

TABLE 4.2 Runtime and peak-memory requirements of LOES and BDD for instances of the airport domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

test domains. As LOES exploits the same redundancies as BDD (albeit to a lesser degree), its compression rate is analogous to the latter, albeit the variance is much smaller. LOES in particular avoids the blow-up BDDs suffer in domains like freecell, microban and the n-puzzles, showing robustness across all test domains. A key advantage of LOES and Packed over BDDs *not evident* in these results is that both allow to easily associate ancillary data to set elements without using pointers, which is necessary for most search algorithms. LOES also represents small sets efficiently making it suitable for many best-first algorithms and/or strong heuristics.

Relative to FD, the results show that the runtime comparison is not in favor of LOES, which took about 10 and 20 times longer than FD on the larger instances both can solve. As shown in the table 4.1, FD uses up to two orders of magnitude more memory than LOES. In fact even on the hardest instances, it generally only ran for less than 5 minutes before either producing a solution or running out of memory. While certain overhead stems from employing LOES, a significant part is due to the testbed implementation used in this comparison. The implementation for example employs a simple successor generator which performs a linear scan over the set of grounded operators to find the ones whose preconditions are satisfied, whereas FD uses a decision tree to quickly determine the set of applicable operators. Another source of overhead is that the implementation is not particularly optimized for bulk insertions and only uses

blocksworld	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Blocks-7-0	38,688	0.13	0.09	0.61	0	2.89
Blocks-7-1	64,676	0.22	0.14	1.23	0	7.45
Blocks-7-2	59,167	0.20	0.13	1.23	0	5.56
Blocks-8-0	531,357	2.60	1.37	9.82	5	64.99
Blocks-8-1	638,231	3.12	1.51	4.91	6	87.77
Blocks-8-2	439,349	2.15	1.13	4.91	4	49.32
Blocks-9-0	8,000,866	43.87	19.13	39.29	90	1832.57
Blocks-9-1	6,085,190	33.37	16.54	39.29	73	1265.1
Blocks-9-2	6,085,190	33.37	15.10	39.29	75	1189.86
Blocks-10-0	103,557,485	629.60	271.02	130.84	MEM	110602
Blocks-10-1	101,807,541	618.96	275.43	283.43	MEM	112760
Blocks-10-2	103,557,485	629.60	283.75	130.84	MEM	110821

TABLE 4.3 Runtime and peak-memory requirements of LOES and BDD for instances of the blocksworld domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

depots	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Depots-3	3,222,296	19.97	2.77	69.81	72	1174.48
Depots-4	135,790,678	1068.38	147.98	924.29	MEM	373273

TABLE 4.4 Runtime and peak-memory requirements of LOES and BDD for instances of the depots domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

driverlog	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Driverlog-6	911,306	3.58	0.81	0.61	21	144.74
Driverlog-4	1,156,299	3.72	0.83	0.61	20	195.22
Driverlog-5	6,460,043	23.10	4.45	1.23	162	1689.65
Driverlog-7	7,389,676	34.36	5.66	1.23	233	2735.61
Driverlog-8*	82,221,721	411.67	64.08	4.91	MEM	228181

TABLE 4.5 Runtime and peak-memory requirements of LOES and BDD for instances of the driverlog domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

freecell	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Freecell-2	142,582	0.97	0.52	9.82	3	80.81
Freecell-3	1,041,645	9.19	4.47	39.29	25	904.13
Freecell-4	3,474,965	36.04	20.19	100.32	95	4321.72
Freecell-5	21,839,155	278.57	128.10	MEM	MEM	53941.8
Freecell-6*	79,493,417	1137.16	519.96	MEM	MEM	481452

TABLE 4.6 Runtime and peak-memory requirements of LOES and BDD for instances of the freecell domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

gripper	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Gripper-5	376,806	1.48	0.11	0.31	3	65.26
Gripper-6	1,982,434	8.74	2.06	0.31	20	466.55
Gripper-7	10,092,510	50.53	2.69	0.61	123	2894.97
Gripper-8	50,069,466	280.53	13.22	0.61	MEM	22720.9
Gripper-9	243,269,590	1479.00	133.92	1.23	MEM	410729

TABLE 4.7 Runtime and peak-memory requirements of LOES and BDD for instances of the gripper domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

microban	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Microban-4	51,325	0.39	0.20	2.46	0.43	9.57
Microban-6	312,063	3.01	1.33	4.91	3.27	247.02
Microban-16	436,656	4.89	2.05	2.46	5	329.95
Microban-5	2,200,488	22.30	10.35	69.81	29.3	676.4
Microban-7	25,088,052	287.11	122.66	741.19	MEM	24574.1

TABLE 4.8 Runtime and peak-memory requirements of LOES and BDD for instances of the microban domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds.

satellite	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Satellite-3	19,583	0.04	0.01	0.04	0	2.25
Satellite-4	347,124	0.95	0.12	0.08	13	74.5
Satellite-5	39,291,149	182.67	14.27	4.91	MEM	28580.9
Satellite-6	25,678,638	97.96	8.27	0.61	MEM	13684.6
Satellite-7*	115,386,375	591.47	37.96	2.46	MEM	380135

TABLE 4.9 Runtime and peak-memory requirements of LOES and BDD for instances of the satellite domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds. Instances denoted by * were still running, in which case the numbers are for the largest layer both BDD and LOES processed.

travel	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Travel-4	7,116	0.02	0.01	0.08	0.08	0.42
Travel-5	83,505	0.22	0.07	0.31	1.46	7.6
Travel-6	609,569	1.82	0.41	1.23	14.4	77.09
Travel-7	528,793	1.58	0.46	0.61	8.45	71.61
Travel-8	14,541,350	62.40	10.28	19.64	MEM	8178.87
Travel-9*	68,389,737	317.95	50.93	161.36	MEM	167795

TABLE 4.10 Runtime and peak-memory requirements of LOES and BDD for instances of the travel domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds. Instances denoted by * were still running, in which case the numbers are for the largest layer both BDD and LOES processed.

mystery	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
Mystery-2	965,838	13.47	3.09	19.64	88.6	469.84
Mystery-4**	38,254,137	228.01	23.52	19.64	MEM	11674.1
Mystery-5**	54,964,076	563.49	150.90	130.84	MEM	290441

TABLE 4.11 Runtime and peak-memory requirements of LOES and BDD for instances of the mystery domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds. Instances denoted by ** have no solution. The numbers hence represent the reachable search-space.

n-puzzle	$\|O \cup C\|$	s_{pck}	s_{loes}	s_{bdd}	t_{fd}	t_{loes}
8-Puz-39944	181,438	0.78	0.42	4.91	1.08	26.49
8-Puz-72348	181,438	0.78	0.43	4.91	1.04	26.37
15-Puz-79*	23,102,481	176.26	102.32	558.08	MEM	17525.6
15-Puz-88*	42,928,799	327.52	188.92	771.70	MEM	71999.7

TABLE 4.12 Runtime and peak-memory requirements of LOES and BDD for instances of the n-puzzle domain. $|O \cup C|$ is the number of states in $Open$ and $Close$ before the first state expansion in the goal layer, s_{pck}, s_{loes} and s_{bdd} are the peak memory requirements (in MB or MEM if the process ran out of it) of Packed, LOES and BDD. t_{fd}, t_{loes} and t_{bdd} the respective runtimes in seconds. Instances denoted by * were still running, in which case the numbers are for the largest layer both BDD and LOES processed.

a small buffer to accommodate new set elements. Whenever that buffer is full, it is combined with the previous LOES representation of the set, which includes iterating over that previous set (which needless to say is costly for the very large layer sets in breadth-first-search). However the focus of this comparison is between LOES and BDD based representations. The testbed allows to run both as black-box configurations in an equal context producing realistic loads. Resulting inefficiencies of the testbed are equally suffered by both techniques (they run after all the same code to generate the same states). In that regard, the results paint a favorable picture of LOES' runtime. Despite the supposedly high overhead from the basic successor generation, it still regularly outperformed BDD in absolute runtime by an order of magnitude.

In conclusion, the results show that LOES offers good space efficiency for representing explicit state-sets of all sizes. It provides these robust space savings even in traditionally hard combinatorial domains such as the n-puzzles where redundancies are relatively hard to exploit. In particular, it does while also defining a consecutive address-space over set elements (i.e. a perfect-hash), which allows space-efficient association of ancillary data to set-elements without addressing overhead.

4.2.14 Pattern Database Representations

Having discussed the basics of LOES and its use in dynamic programming based search algorithm, I now turn my attention to the representation of pattern databases as another relevant use-case for employing memoization techniques in heuristic search (for a short overview of PDBs, see section 4.1.1). For brevity I do not concern myself in detail with pattern selection, domain abstraction and the corresponding regression search, but assume a pattern database has already been computed and exists as some collection of pattern-value pairs. For in-depth coverage of these interesting topics, I would like to point the reader to [HBH+07, HHH07]. First I want to discuss two possible approaches of representing the *pattern-value* mapping as a LOES code.

4.2.15 Combined Layer Sets

The most straightforward representation is to convert all patterns into a LOES code. LOES allows to assign a unique id with every unique pattern in the range $\{0, \ldots, |\text{PDB}| - 1\}$ which can be interpreted as an address (i.e. be used as an offset) of the associated values in a packed bit-string where each record comprises of the minimal amount of bits necessary to discern between the occurring (in the PDB) values. Computation of the heuristic then comprises of determining the id of the pattern using the member-index function (Algorithm 15) and get the

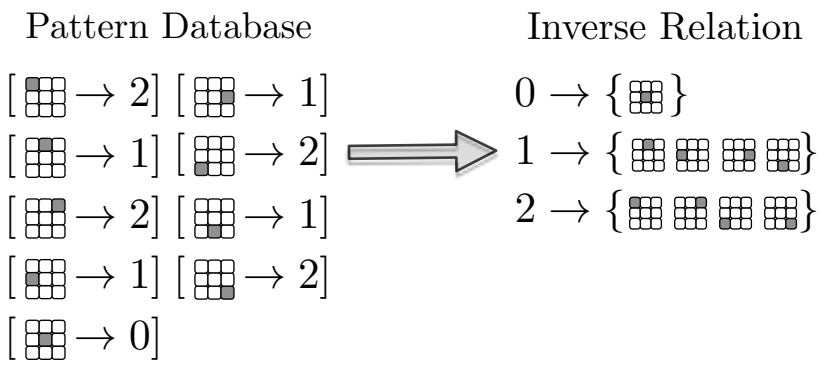


FIGURE 4.25 The PDB for tile 4 of the 8-puzzle and its inverse relation.

value by interpreting id as an offset into the packed bit-string.

4.2.16 Inverse Relation

Especially in unit-cost search, the number of patterns in a PDB usually by far outstrips the number of different values. One can avoid associating these highly repetitive values with individual patterns by storing the inverse of the heuristic function. In general, heuristics are not *injective*, hence a well-defined inverse does not exist. Instead, the *inverse relation* (a left-total relation, where every input is associated with multiple outputs) is stored (see figure 4.25 for an example). That is, each set of patterns sharing a common value is encoded as a LOES and the value associated to the whole set. The heuristic function is then computed through consecutive tests of the pattern against each of the pattern-sets and upon a hit, returning that set's associated value. Note, that due to the function property of the heuristic, these sets are pairwise disjoint. If furthermore, the heuristic is a total function (i.e. the union over all pattern sets comprises the entire abstract pattern space), one can omit storing the largest of the sets and denote its associated value as a default which is returned if the tests against all remaining sets fails. A further optimization is to keep track of the success probabilities of the member-tests over time and query the sets in descending order of these probabilities.

4.2.17 Compressed LOES

Note that for the inverse relation representation, there is no need to associate any information with individual states. The only required operations on the sets are membership tests. This allows for further optimization. The idea is to "mark" inner nodes representing *complete subtrees* (i.e. prefixes for which all possible suffixes are present in the set).If one encounters a root

Algorithm 20: cLOES-PATH-OFFSET

given an encoded state, cLOES-path-offset navigates the cLOES according to the states path interpretation

Input: state a bitsequence of length m
Output: offset an offset into cLOES
Data: cLOES an encoded state-set

```
offset ← 0;
for depth ← 0 to m − 1 do
    if cLOES[offset, offset + 1] = 00 then
        return offset;
    end
    if state[depth] then
        offset ← offset + 1;
    end
    if cLOES[offset] then
        if depth = m − 1 then
            return offset;
        end
        else
            offset ← 2rank_cLOES(offset);
        end
    end
    else
        return ⊥;
    end
end
```

of a complete subtree during a descent through the prefix tree, membership tests can be answered immediately. To exploit this, I developed a variant of LOES, called compressed Level Order Edge Sequence (cLOES), that allows to prune complete subtrees from the structure. The idea is straightforward - I use the remaining code-point 00 (i.e. no further edge at this node) to denote a root of a complete subtree for *inner nodes*. Note that first 00 records do not occur in regular LOES as all records correspond to inner nodes which are guaranteed to have at least a single child and second this does *not* violate the edge-index child-record-position invariant of LOES. As algorithm 20 shows, the changes to member tests are minimal - whenever we reach a new record, we first test if it denotes a complete subtree (i.e. equals 00) and if so return the current offset. Else the algorithm behaves analogously to path-offset. Algorithm 15 can be reused as is, calling cLOES-path-offset instead of path-offset.

Tree construction expectedly differs from LOES. Algorithm 21 gives the procedure analogous to 18 with cLOES modifications (lines 10 to 23). The logic differs as follows. If an insertion generates an 11 record on the leaf level, I convert this to a 00 record. For each such event, I run a simple bottom-up pattern-matching algorithm over (the ends of) all level codes. The pattern to match is where the bit-string on some level i ends in 11 and the bit-string on the lower level in 0000. On a match, I prune the last four bits of the lower level and change the record of the higher level into 11. The intuition behind this is simple - whenever a record sports two edges pointing to complete subtrees, the record is a root-node of a complete subtree.

Figure **??** shows the corresponding LOES and cLOES encodings of a small example set featuring a complete depth 2 subtree. As only inner nodes are represented in a LOES ,this is the smallest configuration that actually leads to a size reduction in the encoding (of 4 bits). The "hook" that allows to derive address spaces over set elements in LOES is that every edge of the prefix tree corresponds to a set bit in the code. These set bits can be efficiently ranked with help of the index structure. Only edges on the last (i.e. lowest) level of the tree can be bijectively mapped to set elements (as all others are potentially shared between members). By pruning edges on the lowest level, cLOES trades this feature for smaller set encodings.

4.2.18 Empirical Comparison of LOES and BDD for Pattern Database Representations

My setup for the following evaluation consisted of a preprocessor for converting PDDL [MGH+98] input files into multivalued problem descriptions similar to Fast Downward's preprocessor [Hel06a]. The difference is that this preprocessor outputs additional *at-most-one* constraints covering the problem variables. They come in the form of lists of variable-assignment tuples

Algorithm 21: cLOES-ADD-STATE

Given a cLOES code partitioned by tree level and a state *lexicographically* greater than the states represented in the LOES, cLOES-add-state manipulates the code partitions, such that their concatenation contains the state (in addition to all prior constituents). It guarantees that complete subtrees are represented as 00 nodes on their root level.

Input: s a bitsequence of length m
Data: codes an array of bitsequences
Data: s' a bitsequence of length m or $\perp$

```
if s' = ⊥ then
    depth ← −1;
end
if s' = s then
    return;
end
else
    depth ← i : ∀j < i, s[j] = s'[j] ∧ s[i] ≠ s'[i];
    if depth = m − 1 then
        codes[depth]]_lastRec ← 00;
        for i ← depth − 1 to 0 do
            if codes[i]_lastRec = 11 ∧codes[i + 1]_last2Recs = 0000 then
                codes[i]_lastRec ← 00;
                codes[i + 1].popRecord();
                codes[i + 1].popRecord();
            end
            else
                break;
            end
        end
    end
    else
        codes[depth].lastBit ← true;
        for i ← depth + 1 to m − 1 do
            if s [i] then
                codes[i] ← codes[i] ∘ ⟨01⟩;
            end
            else
                codes[i] ← codes[i] ∘ ⟨10⟩;
            end
        end
    end
    s' ← s;
end
```

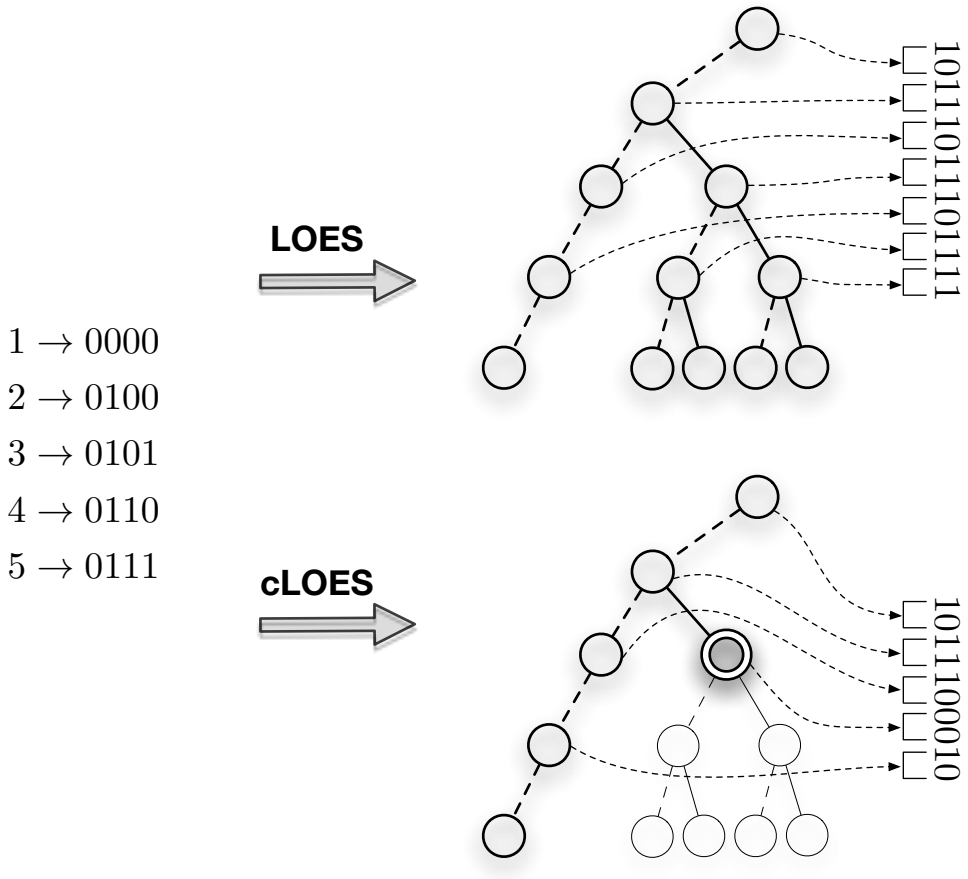


FIGURE 4.26 LOES and cLOES encodings for an example set of five states

and are interpreted such that for any valid state, at most one of the tuples in every list holds true. For the original problem, these constraints add no additional information over what is encoded in the multi-valued description - that is, no successor of the initial state generated through the operator set will violate any of these constraints. They are of use as the original, implicit constraints are often lost when creating an abstraction (i.e. by projecting a problem down to a subset of its original variables).

To give an intuition, consider the multi-valued encoding generated by Fast Downward's (and the above) preprocessor for the N-puzzles. It comprises of one variable for each tile denoting its position. There are operators for every viable pairing of the blank tile and neighboring non-blanks. Each such operator has the specific positions of the blank and the tile as precondition with their switched positions as effect. As tiles start out on different positions in the initial state, the constraint that no two tiles can occupy the same position is implicitly upheld through the operator set. Once even a single tile (i.e. variable) is projected away (which results in the removal of its references from all operator preconditions and effects) that constraint is violated, creating a non *surjective* abstraction (i.e. there are viable patterns in the abstraction, that have no counterpart in the original problem).

This creates two problems. The lesser one is an often exponential increase in the size of the pattern database. The greater one is the severe reduction in quality of the resulting heuristic. If one, say, projects on 7 variables from the 15-puzzle, the resulting database will comprise ≈ 270 million patterns, but as tiles can move "through" each other will carry no more information than sum of manhattan distances (also known as taxicab norm [Kra86]) from the current positions of these 7 tiles to their goal positions. Note, that this does not affect the *admissibility* of the heuristic. Evaluating these "redundant" constraints in the abstract space allows to mitigate this problem by pruning invalid states during the regression search (see also [HBG05]).

The translation process is followed by a rule based system selecting variables for one or more PDBs. Both of these components are experimental at this point which somewhat limit the scope of the following evaluation. These two components derive domains which are abstractions of the original problem. The next step is then to construct the PDBs through a regression search. Here one starts with all goal-states as the initial layer 0 and computes further layers as the set of predecessors (i.e. δ^{-1}) of the last layer minus the content of all prior layers until all reachable states are part of some layer. These layers represent an inverse relationship representation of the PDB as for each state in layer i, it minimally takes i steps to reach a goal in the abstract domain (and hence *at least* i in the original domain). For the comparison, I encoded these PDBs in five representation forms.

PERFECT HASHING (PH) The perfect hash function maps each possible assignment vector (of the abstract problem) to a unique id given by its lexicographic rank. Ids are used for addressing packed records holding the associated values.

BINARY DECISION DIAGRAM (BDD) The PDB is stored as an inverse relation with each set represented as a BDD as described above. Common subgraphs are shared between sets. As above, I used the buddy package, a high performance implementation from the model checking community for my evaluation. If the union of sets comprises all possible patterns, the largest equal-value subset is automatically removed and its value set as default.

LOES Analogous to PH. The perfect hash function is implemented through a LOES set of all *occurring* patterns and its member-index function. If the LOES comprises all possible patterns, the largest equal-value subset is automatically removed from the LOES and its value set as default.

INVERSE RELATION LOES (IR LOES) Analogous to BDD. Each set is represented as a LOES. All sets use the same encoding permutation. If the union of sets comprises all possible patterns, the largest equal-value subset is automatically removed and its value set as default.

INVERSE RELATION COMPRESSED LOES (IR CLOES) Analogous to BDD. Each set is represented as a cLOES with a specific encoding permutation. If the union of sets comprises all possible patterns, the largest equal-value subset is automatically removed and its value set as default.

The PDBs were then used in A^* searches. The Pipesworld Tankage, Driverlog and Gripper instances were run on a 2.2 GHz Intel Core processor running Mac OS $10.6.7$ with 8 GB of memory. For the 15-Puzzle STRIPS instances, a 3.3 GHz Xeon processor with 4 GB of memory was used. The aim was to have the same PDBs represented and used as a heuristic in all five techniques. Hence the employed PDBs had to be relatively small (in the number of patterns) as depending on the domain, the spatial effectiveness of the different representations often differed by multiple orders of magnitude.

4.2.18.1 The Pipesworld Tankage Domain

The IPC-4 Pipesworld Tankage domain models the problem of transporting oil derivatives through pipeline segments connecting areas that have limited storage capacity due to tankage restrictions for each product. The additional constraints made explicit by the preprocessor state

TABLE 4.13 Total PDB size (i.e. number of patterns), solution length and complete search times (parsing, analysis, PDB construction and search) for the Pipesworld Tankage instances.

instance	size	sl	t_{PH}	$t_{\text{IR LOES}}$	$t_{\text{IR cLOES}}$	t_{LOES}	t_{BDD}
Pipesworld-1	67144	5	1.4	1.4	1.7	2.6	3.3
Pipesworld-2	3559	12	0.2	0.1	0.2	0.2	0.2
Pipesworld-3	38204	8	1.8	1.8	2.0	2.1	2.7
Pipesworld-4	85422	11	5.0	4.9	5.7	5.9	7.4
Pipesworld-5	212177	8	9.9	10.2	10.8	11.6	16.7
Pipesworld-6	113364	10	9.0	9.3	9.8	9.8	11.6
Pipesworld-7	13620	8	4.2	4.3	4.1	4.2	4.7
Pipesworld-8	35307	11	11.5	14.4	11.9	12.8	16.6

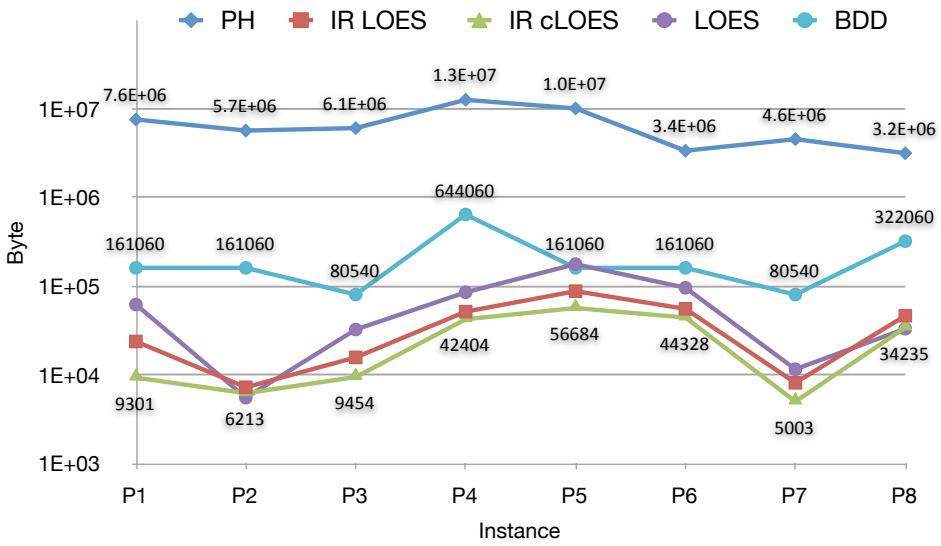


FIGURE 4.27 Size of the PDB representations in bytes for the Pipesworld Tankage instances.

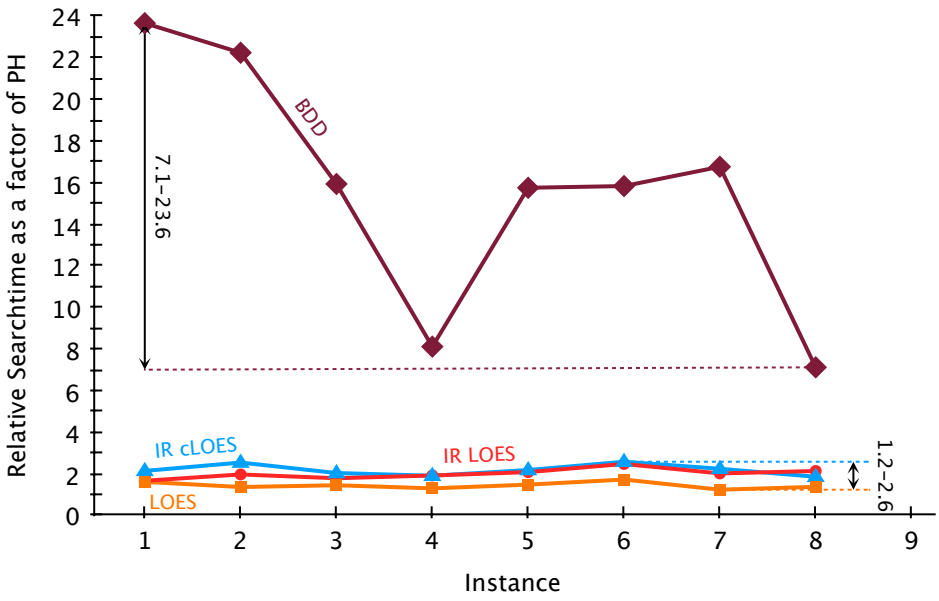


FIGURE 4.28 Relative search time as a multiple of PH for the Pipe Tankage instances.

that for any pipe, there can only be one batch that is closest to a source area and one batch that is closest to a destination area. The analysis component generated single PDBs for all instances. The PDBs are relatively small and retain a good amount of the original problem's constraints. This shows in the sizes for the different representations (see Figure 4.27) where BDD outperforms PH by between one to two orders of magnitude, with the LOES versions besting this by another order of magnitude.

On the time dimension (see Figure 4.28), LOES only performs marginally worse than PH while the IR variants take about twice as long. BDD performance varies considerably and performs a good order of magnitude worse than PH and the LOES encodings.

4.2.18.2 The Driverlog Domain

The IPC-3 Driverlog domain involves delivering packages between locations using trucks. The complication is that the trucks require drivers who must walk between trucks in order to drive them. The paths for walking and the roads for driving form different maps on the locations.

Driverlog is an example where the preprocessing fails to uncover any explicit constraints over those encoded in the multi-valued variables. This results in PDBs of very low quality comprising of all possible abstract patterns. It is also a domain that is quite amendable to BDD representation. This shows in the space comparison (see Figure 4.29), where the BDD

TABLE 4.14 Total PDB size (i.e. number of patterns), solution length and complete search times (parsing, analysis, PDB construction and search) for the Driverlog instances.

instance	size	sl	t_{PH}	$t_{\text{IR LOES}}$	$t_{\text{IR cLOES}}$	t_{LOES}	t_{BDD}
Driverlog-1	30375	7	0.3	0.4	0.6	0.6	0.8
Driverlog-2	339750	19	8.9	9.2	11.2	12.6	16.3
Driverlog-3	1766250	12	37.4	35.7	38.5	51.3	75.2
Driverlog-4	2466625	16	125.5	118.0	120.7	151.9	179.4
Driverlog-5	1800000	18	54.9	55.5	57.1	65.6	83.5
Driverlog-6	4478976	11	266.5	267.7	267.3	294.2	352.0
Driverlog-7	3779136	13	333.1	337.7	334.7	359.4	417.9

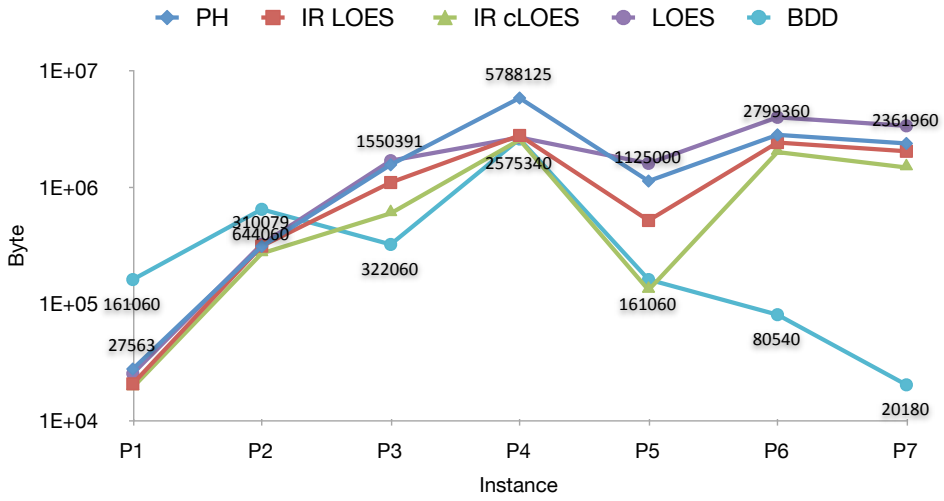


FIGURE 4.29 Size of the PDB representations in bytes for the Driverlog instances.

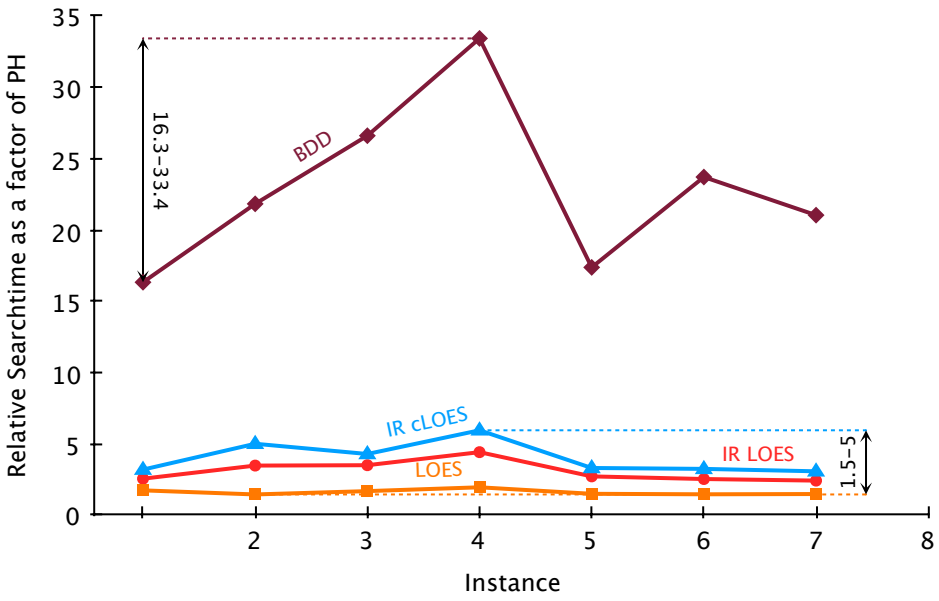


FIGURE 4.30 Relative search time as a multiple of PH for the Driverlog instances.

shines on the large, multi-million pattern instances (that are scarcely subdivided by the IR representation), actually taking up an order of magnitude less space than on the smaller instances. Remarkably the IR LOES variants still manage to outperform PH by a factor of two to three. LOES predictably performs worse as its packed store alone is nearly the same size as PHs (the difference stems from the fact, that it can omit storing the largest equal-value subset of patterns in PBD and denote the corresponding value as its default).

The runtime comparison (see Figure 4.30) paints a similar picture, as the representations' look-up cost is similar to the Pipesworld Tankage instances.

4.2.18.3 The Gripper Domain

The IPC-1 Gripper domain models a mobile robot that can use its two grippers to pick up and put down balls, in order to move them from one room to another.

In this domain, the preprocessor picked up the implicit constraint that no object can be in both grippers at the same time. The variable selection logic constructed PDBs comprising of the variables for the gripper states, the location of the robot and goal qualified balls. A rule was in place that would split PDBs as the abstract state space grew too large. The resulting PDBs were not additive and were hence combined by taking the maximum of their heuristic values. The preprocessing logic opted for multiple PDBs beginning with instance 6, mitigating the growth of the PDBs (see Figure **??**).

instance	size	sl	t_{PH}	$t_{\text{IR LOES}}$	$t_{\text{IR cLOES}}$	t_{LOES}	t_{BDD}
Gripper-1	600	11	0.0	0.0	0.0	0.0	0.0
Gripper-2	4604	17	0.0	0.1	0.1	0.1	0.2
Gripper-3	30320	23	0.4	0.6	1.2	1.0	1.4
Gripper-4	181428	29	3.6	3.9	8.1	7.5	9.9
Gripper-5	1016072	35	28.5	25.9	37.3	52.3	63.6
Gripper-6	1460128	41	51.7	63.7	78.2	90.4	190.5
Gripper-7	1975008	47	136.6	237.8	277.8	206.3	746.3
Gripper-8	2582788	53	574.9	1187.4	1346.0	757.7	3751.0

TABLE 4.15 Total PDB size (i.e. number of patterns), solution length and complete search times (parsing, analysis, PDB construction and search) for the Gripper instances.

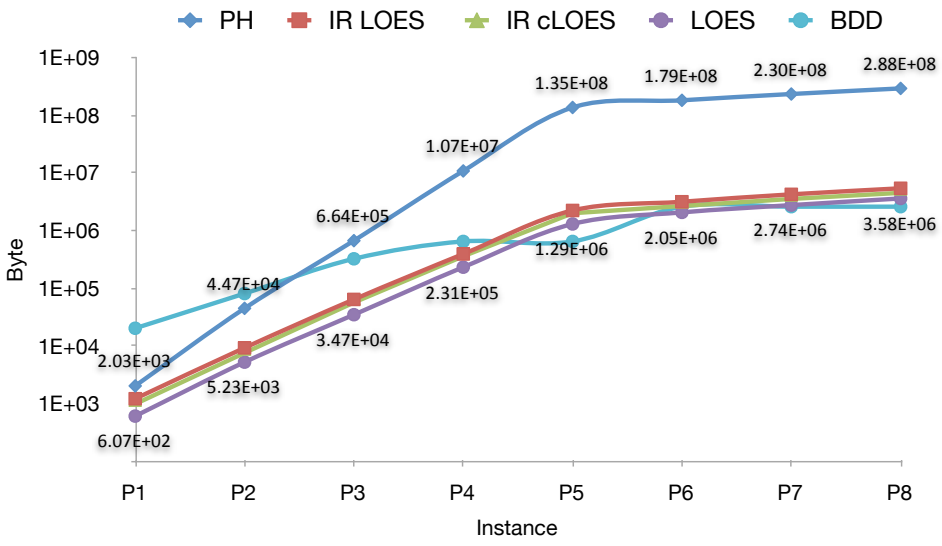


FIGURE 4.31 Size of the PDB representations in bytes for the Gripper instances.

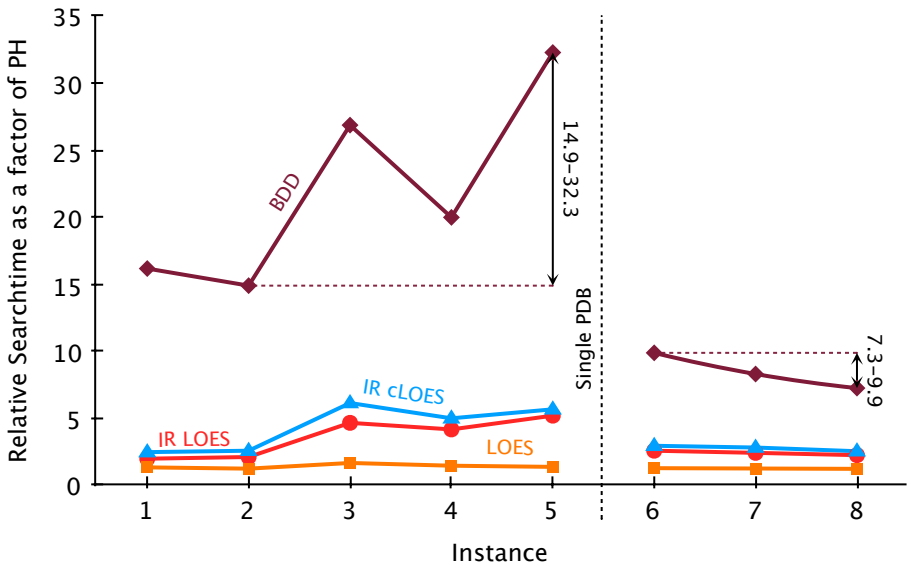


FIGURE 4.32 Relative search time as a multiple of PH for the Gripper instances.

Gripper is one of the domains where BDDs are known to perform extremely well. Still it outperformed IR cLOES in storage efficiency only in instances 5 and 8, when the PDBs where about 1 and 2.6 million patterns in size. PH consistently required around 2 orders of magnitude more storage on the larger instances. The runtime comparison (see Figure 4.32) paints an interesting picture. For the smaller PDBs, PH is about 1.3 (LOES) to 5 (IR cLOES) times faster than the LOES versions. As the pattern databases grow the advantages from the quick addressing shrink considerably, probably due to increased cache misses. Again the more complex matching in BDDs is a good order of magnitude slower.

4.2.18.4 The 15-Puzzle Domain

The 15-Puzzle is a classic combinatorial search benchmark. It is also a token problem for pattern database heuristics (e.g. [CS98]). The preprocessor works quite well in this domain and manages to extract constraints ensuring that no two tiles can occupy the same position. Furthermore the analysis component manages to create *additive* PDBs by excluding the blank and selecting tiles up to its pattern space size limit (which allowed up to 6 variables in this domain), resulting in an additive 6-6-3 PDB. While this represents an unusually strong PDB for an domain-independent planner, it still noticeably weaker than the handcrafted and blank-compressed additive PDBs typically employed in domain-specific n-puzzle solvers (e.g. [FKH04]). I run the planner over Korf's 100 random instances [Kor85], which are known

to be quite challenging for domain-independent planners (e.g., the state-of-the-art Fast Downward planner using a merge & shrink heuristic with an abstraction size of 10^4 nodes cannot solve the easiest instance within 96 GB of RAM). It is also a permutation problem and hence a hard domain for the type of redundancy-elimination techniques employed by LOES and particularly BDDs. As the analysis component is deterministic, the same pattern database was

TABLE 4.16 Number of instances solved, PDB size and relative search speed over Korf's hundred 15-puzzle instances.

Rep.	$\#_{solved}$	size (MB)	$\frac{t_x}{t_{PH}}$
PH	91	20.0	1.00
BDD	40	117.6	0.11
IR LOES	82	11.4	0.36
IR cLOES	70	9.6	0.23
LOES	68	11.2	0.49

generated for all instances. Table 4.16 shows the resulting sizes for the different representations. Despite the permutation problem, LOES' succinct encoding ends up around half the size of PH. While this represents a noticeable relative reduction in PDB size, the absolute differences were too small to significantly impact the A^* runs. The BDD based representation was predictably not competitive. I ran all instances with a hard 30 minute cut-off timer. Figure 4.33 gives the total planning time per instance (i.e. parsing, analysis, PDB creation, -encoding and the A^* search) and figure 4.34 the pure search time. As all PDBs are qualitatively equal, PH fared best with 92 solved instances, thanks to its simple and fast lookups. Plain LOES fared worst of all LOES variants despite its fast search speed (at $\approx 49\%$ of PH), as its PDB representation is the most time-consuming to build. These results would change if the problem were memory-bottlenecked, e.g. if on a 25-puzzle the analysis component allowed PDBs of such size, that the additional space overhead of PH would represent a bottleneck for the A^* algorithm.

An important point to keep in mind is that for pattern databases, all encodings map well to a common interface. The question for a domain-independent planner is more or less which representation to choose. In all, this comparative evaluation was arguably limited by the employed experimental preprocessing and analysis components which in most domains I tried, failed to extract additional constraints that help in defining informative abstractions. I have included Driverlog as a representative of these cases. As abstraction selection rules and heuris-

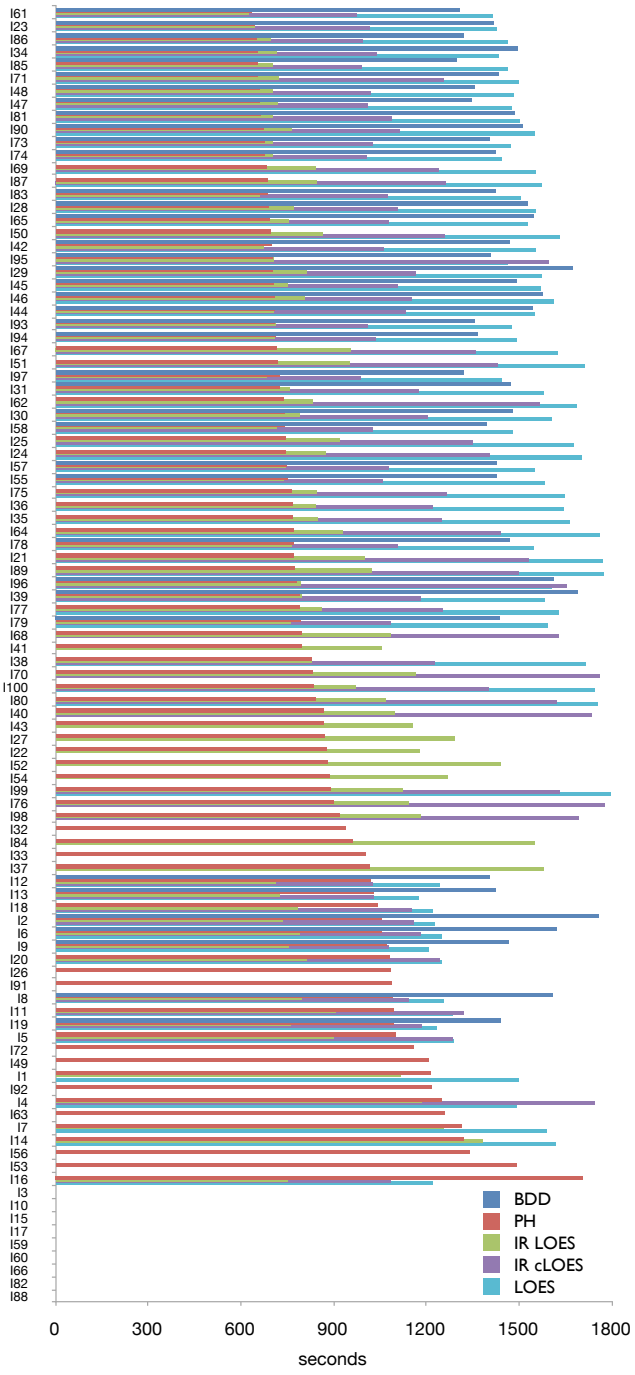


FIGURE 4.33 Planning time (parsing, analysis, PDB creation and search) over Korf's 100 instances of the 15-puzzle in ascending order of duration for PH. I23 stands for instance 23. If a representation has no bar for an instance, it failed to terminate successfully in 30 minutes.

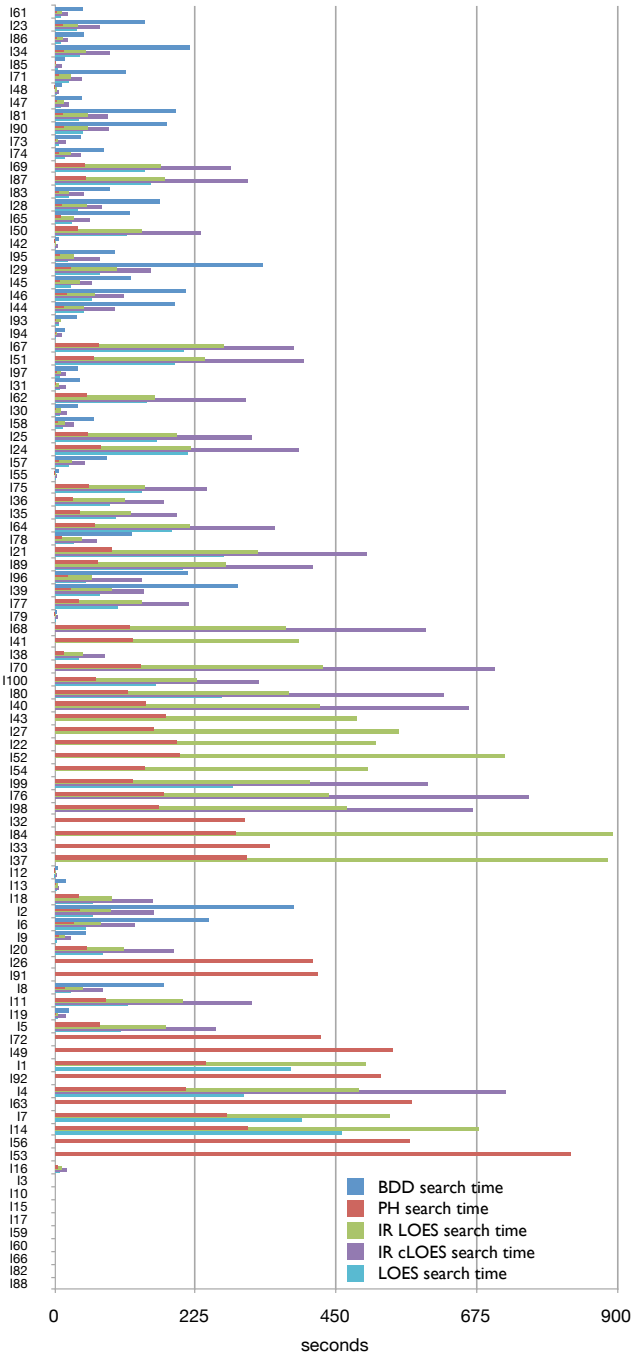


FIGURE 4.34 Search time over Korf's 100 instances of the 15-puzzle in the order corresponding to Figure 4.33. If a representation has no bar for an instance, it failed to terminate successfully in 30 minutes.

tics evolve and ripen, it is quite probable that such unconstrained (i.e. the number of patterns in the PDB equals the product of the patterns' variables' domains) and weak PDBs (i.e. few different values) are eventually generated. While generally undesirable as they take up large amounts of memory while offering little guidance, there is a good chance as the Driverlog example shows, that a BDD-based representation can reduce the size by orders of magnitude and make the trade-off worthwhile. In domains where the selection process manages to derive good quality PDBs such as Pipesworld Tankage, Gripper and n-puzzles, the LOES variants usually show the best spatial efficiency. Somewhat surprisingly that even extends to small PDBs in permutation domains (see the n-puzzles). BDDs are competitive, if the domain is very amendable to its redundancy reduction techniques (i.e. Gripper). On the time dimension, the picture is much clearer. First, all three techniques have the theoretical advantage, that the computational complexity of heuristics lookups only depend on pattern complexity and not on PDB size. Second, the LOES variants are about 1.5-5 times slower than PH and the BDD adds another order-of-magnitude on top of that. As PDBs grow to more reasonable sizes PH's advantage shrinks considerably. While LOES and BDD require multiple memory accesses per lookup, their hierarchical structure and small encodings are good fits for current machines memory hierarchy. PH on the other hand can make do with a single sequential read but the corresponding accesses are uniformly distributed over a large memory range (i.e. mainly due to its worse spatial efficiency) with a comparably larger chance of cache misses. This effects already manifest themselves on the larger test PDBs (see figures 4.32,4.28 and 4.30). Note however that in many domain-dependent applications, PDBs often comprise in the hundreds of millions of states, probably strongly amplifying this effect. Finally in permutation problems with relatively small PDBs, PH is competitive. Particularly if the computation is time limited, the tradeoff of PH's faster lookups for its slightly worse space efficiency may be well worth it as the n-puzzles example shows.

4.3 Summary

I began this chapter with an overview of common representation techniques used in state-of the art propositional planning and from there introduced LOES, which I believe offers exciting opportunities to better exploit dynamic programming and other memoization techniques in domain-independent planning. It shows good space efficiency for representing explicit state-sets of all sizes and provides robust space savings even in traditionally hard combinatorial domains such as the n-puzzles. In particular, it defines a consecutive address-space over set elements, which allows space-efficient association of ancillary data to set-elements without ad-

dressing overhead which makes it compatible with a wide range of search algorithms. LOES also allows for quite efficient representation of strong, precomputed heuristics. The very basic domain analysis I employed in the above evaluation can only give a hint of the potential for ad-hoc abstraction in heuristic search. Particularly noteworthy is also LOES' impedance match with BDDs and PH for PDB representations. The inverse relation representation straightforwardly allows to adaptively interchange BDD and LOES based representations of state-sets. In this way, a domain-independent planner can leverage the superior efficiency of BDDs in appropriate domains while mitigating their lack of robustness by falling back to a LOES-based representation. In all, I am convinced that techniques such as LOES are a worthwhile addition to the toolkit of propositional planning.

CHAPTER 5

Parallelization of Heuristic Search

From its beginnings in the 1960s to today, the scale of problems that can be reasonably tackled by heuristic search has increased immensely. Next to algorithmic advances this was due to an exponential increase in computational capability per dollar thanks to advances in semiconductor design and manufacturing over the last six decades. Moore noted in 1965 that the average number of components in an integrated circuit doubled every 18 months and predicted that trend would continue "for at least ten years" [Gor65]. This prediction has more or less held to this day and became known as "Moore's Law". CPU designers leveraged these advances and the accompanying possibility to increase operating frequency resulting in an equally exponential growth in processing speed and hence provided vast "free" speed-ups of heuristic search algorithms (and other programs). In the mid 2000s, power consumption more and more emerged as the main roadblock preventing the continuation of this trend in CPU design. As roughly, power consumption is linear in the number of components and performance increase of a CPU due to architectural advances is roughly proportional to the square root of the complexity increase (known as "Pollack's Rule"), to further exploit manufacturing advances the design trend changed to integrating more and more processors on a single integrated circuit.

5.1 Background

One recently popular approach to exploiting parallelism in planning is to run parallel executions of different heuristic search algorithms (or, often simply different heuristics) over one given problem (e.g. [HRK]), The efficiency of state of the art heuristics and algorithms varies widely with a problem's underlying domain structure. In scenarios where the task is solve a single or small number of problem instances without in-depth knowledge over their underlying domain, this idea exploits that typically for any given problem, some class of heuristics vastly outperforms its peers. The general idea has long been popular in many hard, computa-

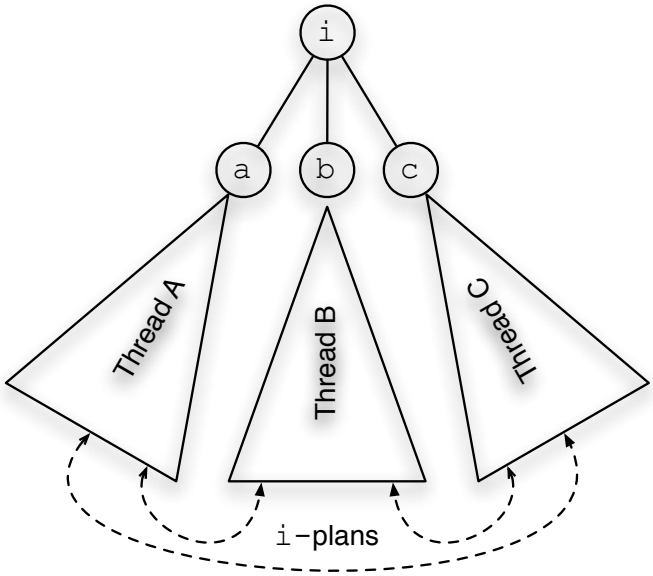


FIGURE 5.1 In Parallel Depth-First Search, a short breadth-first search is run until the size of the frontier matches or exceeds the desired number of threads. A DFS task is then spawned for state in the frontier. The only interdependency between these tasks is the necessary dissemination of (task-local) i-plans.

tional problems. Huberman et al. [HLH97] give a general framework for principled selection of participants for such runs (or portfolios) based on risk theory.

The following discussion however is based on a different scenario. Here I assume that the problem domain is understood to a degree that allows the selection of a suitable heuristic and algorithm combination and the intent is to better exploit the inherent parallelism of modern CPUs to solve larger instances in such a domain. A direct result of this scenario is that adaptations have to be made on the algorithmic level to parallelize the selected search algorithm.

For depth-first algorithms, these modifications are often trivial. For example, creating DFS tasks for all elements in $\delta(i)$ is a straightforward and efficient modification. The only necessary shared state between these tasks is the current best solution. This value needs only be read (and potentially be updated) if a participating thread uncovers a new i-plan. While DFS is hence technically not an *embarrassingly parallel* workload as intermediate results have to be communicated between tasks, these communications are so rare that synchronization with platform primitives is not a bottleneck. This basically provides linear speed-up for domains suitable to depth first search (see [RK87] and [KR87] for a discussion of parallel DFS on amongst others the n-puzzles). Things are different for Best-First Search with duplicate detection. To guar-

antee admissibility, states must be expanded in order of their increasing f-estimates across all threads. To avoid redundant expansions, every previously unknown state generated by one task must be broadcast to all other tasks immediately. State-of-the-art heuristic-search planners such as Fast Downward [Hel06a] generate on the order of hundreds of thousands of states per second in most domains on a contemporary CPU. Platform level synchronization primitives such as semaphores do not scale to such levels - in fact the computational overhead is such that even in shared-memory systems and *regardless* of the number of involved CPUs, straightforward adaptations of A* (e.g. ensuring mutual exclusive access to shared $Open$ and $Close$ structures) often give *reduced* performance when compared to their single-threaded (i.e. mutex free) counterparts in most domains, sometimes by orders of magnitude as shown in [BLZR09].

More significant changes to the control-flow of BFS algorithms are needed to provide the necessary scalability. As this is a relatively novel area of heuristic best-first search, the corresponding body of work is rather manageable. The mutual toehold is to partition $Open$ and $Close$ in a way that reduces synchronization overhead. One thread of research has focused on using hash-functions to assign states to search tasks. Every state generated is immediately run through the hash-function and assigned a task id. If the id does not match the generating task, the state is transferred accordingly. Each task holds local $Open$ and $Close$ collections for states with its id and is solely responsible for duplicate detection and expansions of such states. Hence none of these structures need to be globally synchronized. Analogous to the modified A* in section 3.3.1, the first solution discovered by a subtask is not necessarily optimal - it can only be guaranteed if the f-value of the current best global solution is less or equal than the smallest f-values in every task's $Open$ queue. This scheme forms the basis of Parallel Retracting A* (PRA*) [EHMN95] which was specifically developed in the early 90s for the *connection machine* series of supercomputers. PRA* also differs from vanilla A* in that it allows the retraction (hence the "R" in the name) of expanded nodes to preserve memory. Conceptually, retraction replaces the children of some node n with f-estimates higher some value c with a $\langle n, c \rangle$ tuple in $Open$ to reduce its size. The tuple can later be re-expanded as the children move into the frontier (for details see, again, [EHMN95]). Relative to the computational abilities of its processors, the connection machine architecture is characterized by very high bandwidth and low-latency inter-processor computations (see [Kat87]). Despite this the authors note that over-congestion of the inter-cpu communication network usually limited scalability of the algorithm, a somewhat expected result as nearly every generated state needs to be moved to another task (i.e. CPU). On basically all contemporary computing platforms, the relative (in CPU cycles) latency of sending structured data between tasks/processes

is much higher. Hash-Distributed A* (HDA*) [KFB09] adapted PRA* to this effect by using buffered, asynchronous communications between tasks which somewhat alleviates the latency issue. The bandwidth issue however still remains. On n CPUs, every state generated by a task must be moved to another task with probability $^{n-1}/_{n}$ (under the assumption of a perfectly distributing hash unction). As $\lim_{n\to\infty}(^{n-1}/_{n}) = 1$, basically every state *generated* (including *all* duplicates) by some processor in HDA* must be moved over the inter-processor connection network when running in a highly parallel regime. In the empirical evaluation by [KFB09], this results in a significant drops of scaling efficiency at around four cores on a high-end off-the-shelf workstation and around 64 cores on the TSUBAME[1] grid cluster (see [KFB09]).

5.1.1 Parallel Structured Duplicate Detection

The above approaches use hash-functions (e.g. [Zob70]) on the (SAS+ in the case of (HDA*)) encodings with the aim to distribute generated states as evenly as possible amongst participating tasks. While the task-local partitioning of $Open$ and $Close$ avoids synchronization overhead, complete disregard for the consequential communications overhead cripples their performance on anything but special-purpose hardware. Very similar challenges arise for duplicate detection in external memory search, where the assumption is that large parts of $Open$ and $Close$ reside in comparatively slow, block-oriented memory. Here Zhou and Hansen [ZH04b] showed that partitioning the search graph based on an abstraction of the state space and expanding nodes in the frontier in an order that respects this partitioning can drastically limit the necessary number of costly I/O operations and increase overall performance. Such abstractions are straightforward to generate, by projecting a planning instance to a subset of its propositional variables. As an example consider the projection of apartment domain instances (c.f. section 2.1.2) to $P' = \{p_A, p_B, p_C, p_D\}$. All references of variables in $P \setminus P'$ in O, i and G are removed to form the respective O', i' and G' (see section 4.1.1 for another example).

The abstract domain induces the much simplified search graph given by figure 5.2. Note that the abstraction preserves successor relationships. That is, for every pair of domain states s, s' s.th. $s' \in \delta(s)$ in apartment, there is a correspondingly labeled transition between the abstractions $a = abs(s)$ and $a' = abs(s')$ in the abstract search graph. Or in a slightly different interpretation, the successors of a state mapping to a under abstraction abs map to $\delta'(a)$. Hence by partitioning $Open$ and $Close$ according to their elements respective abstract states, duplicate detection of successors of a state s can be limited to partitions pertaining to $\delta'(abs(s))$. This set of partitions is called the *duplicate detection scope* of s (see figure 5.3 for

[1] a supercomputer that ranked the 7-th fastest in the world when it was installed in 2006

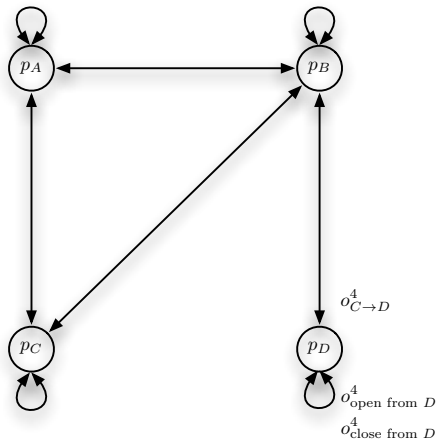


FIGURE 5.2 Domain state transition graph for the projection of the apartment domain to $\{p_A, p_B, p_C, p_D\}$. As these propositional variables adhere to an *exactly one* constraint in apartment, I denote abstract states by the corresponding $true$ propositional variable. The figure only gives operator labels for transitions ending in p_D.

an example). The concept is useful for external memory search as under a suitable abstraction, these partitions are small enough, that during expansion, a partition of the frontier and its corresponding duplicate detection scope can be held in RAM, significantly speeding up the search.

The same authors later leveraged the technique for parallel breadth-first heuristic search [2] [ZH07b]. The idea of *parallel structured duplicate detection* (PSDD) is straightforward - partitions of the frontier with *disjunct* duplicate detection scopes can be expanded in parallel without any need for further synchronization. As synchronization is now done on the level of (rather large) sets of states at a low frequency, its overhead is usually negligible. Later Burns et al. applied PSDD to optimal [BLZR09] and approximate [BLRZ09] best-first search with good success.

5.2 Parallel Edge Partitioning

One potential shortcoming of PSDD (or SDD in general) is its dependence on local structure in the abstract search graph. First, a helpful definition.

Definition 1. *The* maximum concurrency *of parallel search is the maximum number of parallel processes allowed during search such that no synchronization is needed for concurrent*

[2] in BFHS a heuristic is used to prune $Open$, not to guide the search.

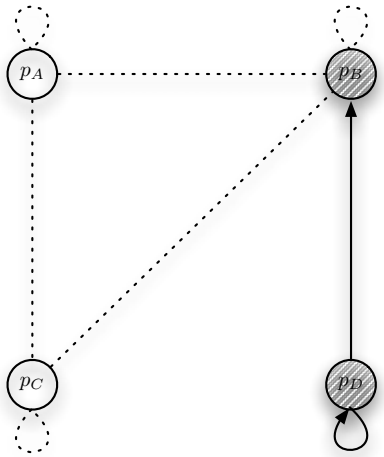


FIGURE 5.3 Duplicate detection scope $\{p_C, p_D\}$ for expanding states corresponding to abstract state p_D in apartment. The scope comprises of states pertaining to those abstract states that have an incoming transition from p_D.

node expansions in these processes.

Lemma 1. *The maximum concurrency of parallel structured duplicate detection or parallel edge partitioning under some abstraction is the maximum number of disjoint duplicate-detection scopes in the corresponding abstract state-space graph.*

For the apartment example (see figure 5.2), maximum concurrency according to lemma 1 is 1, i.e. no two abstract states have disjunct duplicate detection scopes. In fact, the scope of p_C is the complete state space. The abstract graph has little *local structure* mostly because the domain is exceedingly small. In the worst case, domain structure is such that no abstraction results in any concurrency (i.e. any abstract graph is fully connected). Figure 5.4 gives a simple single-tile abstraction of the 15-puzzle domain resulting in a maximum concurrency of 4. More elaborate abstractions, i.e. the position of two or three tiles can raise this significantly. Note however that the abstract domain graph usually needs to be kept in memory and can amount to significant spatial overhead for very fine-grained abstractions. Also as the average amount of states per partition in the frontier decreases, relative synchronization overhead increases accordingly. In all SDD depends heavily on a suitable domain abstraction.

Parallel Edge Partitioning (PEP) avoids these issues by using a staggered expansion scheme. Each expansion task is associated with an edge (a, a') of the abstract domain graph in that only the subset of operators corresponding to said edge are applied during the expansion. This leads to the following definition.

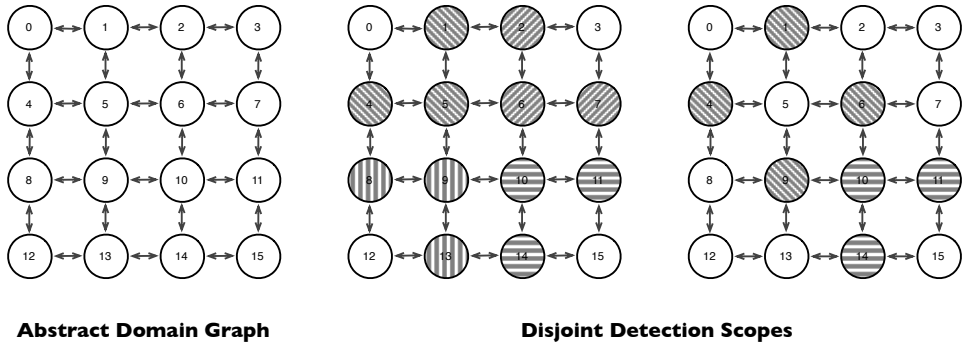


FIGURE 5.4 Domain state transition graph for the abstraction to the position of a single tile of the 15-puzzle domain. Abstract states are denoted by the possible positions 0 to 15. Maximum concurrency for this abstraction is 4, as the duplicate detection scope of positions 0,3,12 and 15 show. The figure on the right shows that concurrency is a factor of the selected abstract states.

.

Definition 2. *An* edge-partitioned duplicate-detection scope *of a state s with respect to an abstract edge (a, a') under a state-space projection function abs s.th. $a = abs(s)$ corresponds to the set of stored nodes that map to abstract node a'.*

As with SDD, the edge-partitioned duplicate-detection scope of a state s with $abs(s) = a$ under the subset for operators pertaining to (a, a'), i.e. the subset of all states in $Open$ and $Close$ mapping to a' under abs) is guaranteed to contain all potential duplicates of s' successors. As structured duplicate detection, edge partitioning was originally devised for external memory search (see [ZH07a]). In this context it is important during expansion that the states pertaining to any abstract node and its duplicate detection scheme fit in memory. For an abstraction to P', each abstract node can represent a set of up to $2^{|P \setminus P'|}$ states in $Open$ and $Close$ (i.e. all possible combinations of the propositional variables abstracted away). To guarantee this abstraction choice has to be constrained to cases where representing $(n+1)2^{|P \setminus P'|}$ (with n denoting the maximum out-degree in the induced abstract graph) states does not exceed available memory.. However this is a weak bound in practice as the average out-degree is often much smaller. With edge partitioning, the analogous bound is $2^{|P \setminus P'|+1}$ and hence independent of structural properties of the abstract graph. In practice, this allows external memory search to scale to significantly larger problems (see [ZH07a] for empirical evaluation of SDD and EP on a variety of IPC domains). In the context of parallel search, given the same abstract search graph, parallel edge partitioning intuitively allows for a significant increase in maximum concurrency as figure 5.5 shows for the two examples. Taking a more principled view on the issue, the following holds for structured duplicate detection.

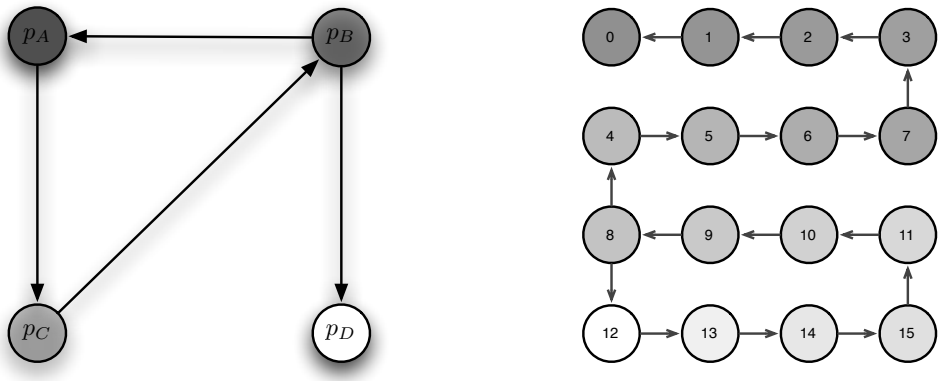


FIGURE 5.5 Maximum concurrency (4 and 16 respectively) with parallel edge partitioning for the abstractions of the apartment and 15-puzzle domains. Note that multiple tasks can be expanding the states of the same abstract set in parallel as long as the target sets are disjunct (e.g. p_B in apartment and 8 in 15-puzzle).

Theorem 1. *The maximum number of disjoint duplicate-detection scopes under structured duplicate detection (without edge partitioning) is bounded by the size of the abstract state-space graph divided by the minimum out-degree of the abstract graph, that is,*

$$\left\lfloor \frac{\textit{size of abstract graph}}{\textit{minimum out-degree of abstract graph}} \right\rfloor$$

Proof: An abstract node with the minimum out-degree has the fewest number of successors, which in turn produces the smallest duplicate-detection scope (in terms of the number of abstract nodes). Suppose an abstract graph can be partitioned into k disjoint scopes. Since the same abstract node cannot appear in more than one scope, the value of k cannot exceed the total number of abstract nodes divided by the size of the smallest scope or, equivalently, the minimum out-degree. □

Corollary 1. *The maximum concurrency of parallel structured duplicate detection (without edge partitioning) is bounded by the size of the abstract state-space graph divided by the minimum out-degree of the abstract graph.*

Corollary 2. *The maximum concurrency of parallel structured duplicate detection (without edge partitioning) on a fully connected abstract state-space graph is one (i.e., no concurrency).*

From the above theorem, these two corollaries can be deduced directly. In practice for abstractions of reasonable coarseness, dividing the number of states in the abstract graph by their

average out-degree usually gives a reasonable benchmark for the supported concurrency of the abstraction. The second corollary shows that PSDD can (as alluded earlier) fail to extract any synchronization-free, parallel expansion regime even when the abstraction is non trivial (i.e. there is more than a single abstract node in the graph).

Theorem 2. *The (maximum) number of disjoint duplicate-detection scopes under edge partitioning is the size of the abstract state-space graph.*

Corollary 3. *The maximum concurrency of parallel edge partitioning is equal to the size of the abstract state-space graph.*

Theorem 2 and corollary 3 follow straightforwardly from the property that each duplicate detection scope under PEP corresponds to a single abstract node. In contrast to SDD, PEP is much more likely to operate close to its respective concurrency bound during layer expansion (i.e., as figure 5.4 shows, maximum concurrency can not be sustained during expansion of some abstract nodes in SDD). Note however, that in practice some abstract nodes usually correspond to empty sets in the frontier and non-empty sets vary greatly in size.

Theorem 3. *For any given state-space projection function, the maximum concurrency of any parallel search algorithm is bounded by the size of the abstract state-space graph if duplicates must be detected as soon as they are generated.*

Proof: Suppose the size of the abstract graph is k and yet the maximum concurrency of the parallel search is greater than k. Without loss of generality, assume there is a $(k+1)$-th process that can join the other k processes with no synchronization. Since duplicates must be caught as soon as they are generated, this means the $(k+1)$-th process must be given exclusive access to its duplicate-detection scope, which consumes at least one abstract node. But since there are only k abstract nodes, according to the pigeonhole principle, there must be one abstract node that is shared by two processes and they must synchronize with each other to perform duplicate detection simultaneously. This leads to a contradiction, which proves the $(k+1)$-th process must not exist, and instead the Theorem must hold. □

5.2.1 Parallel Breadth-First Search with Duplicate Detection and Edge Partitioning - an Integration Example

Having discussed the basic principle of parallel edge partitioning, I now want to show how the technique can be used to easily parallelize a search algorithm. For clarity, I chose the straightforward BFS-DD as my example case but the approach transfers naturally to other

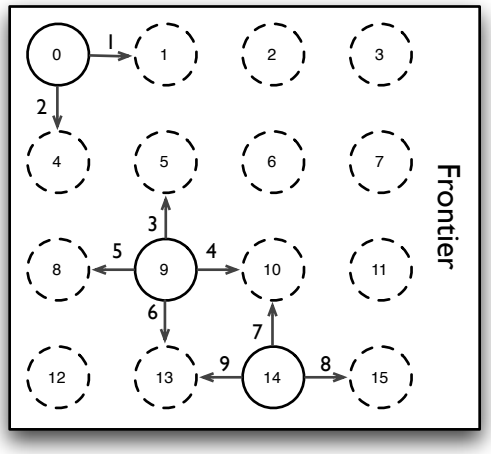


FIGURE 5.6 Example of a search frontier for a 15-puzzle instance partitioned according to position of the blank tile. Dotted partitions are empty. Jobs (1 through 9) are created corresponding to each out-going edge of a non-empty partition (0, 9 and 14) in the frontier.

unit-cost heuristic best-first search algorithms. As the actual sequential algorithm has already been given and discussed thoroughly in section 2.3.2 I will concentrate on the mutual synchronization aspects of parallelizing the algorithm in the context of a common threading package[3]. The overall scheme is relatively straightforward. I assume the runtime environment provides a pool of k worker threads representing available parallel execution units. W.l.o.g. thread 1 is designated the coordinator of the group. I assume that $Open$ (comprising of two layers frontier and next) and $Close$ are partitioned according to some abstraction abs. Furthermore I assume that the corresponding abstract search graph has been generated in a preprocessing step and is readily available in memory with its edges labeled with the sets of corresponding operator ids.

Listing 22 gives the algorithm run on all participating threads with the exception of the initialization code between lines 1 and 2 that I included for completeness. All threads are synchronized to stay within a single layer through an initial barrier (line 3). Before the layer expansion, a job list is generated by the coordinator thread. To this end it first scans the frontier for non-empty partitions and creates a job for each outgoing edge of these partitions (line 4). Jobs are four-tuples comprising of the source partition id, target partition id, the set of operator-ids corresponding to the abstract transition and the job's current progress in its lifecycle (comprising of discrete states $READY$, $PROCESSING$, $FINISHED$). Figure 5.6 gives an example for a simple abstraction in the 15-puzzle.

The thread then sets the binary variables associated with each partition to denote their avail-

[3] modeled roughly after the BOOST thread library

Algorithm 22: PAR-BFS-DD

Parallel Breadth-first search with duplicate detection;

Output: an optimal i-plan or $\perp$

```
1  // Initialization by the calling context
   if i ∈ G then
       return [];
   end
   next ← {i};
   // solution is a synchronized variable
   solution ← ⊥;
2  // Spawn threads with the following algorithm
   while true do
3      Barrier;
       // Sequential layer initialization
       if id = 1 then
           Close ← Close ∪ frontier;
           frontier ← next;
           next ← ∅;
4          extract-jobs-from-frontier( job-array);
           if job-array = ∅ then
               Terminate all threads;
           end
5          ∀abstract-node : abstract-node.free ← true
       end
6      Barrier;
       // Parallel layer expansion
7      while ∃job ∈ job-array s.th. job.state = READY do
8          if atomic-test-and-set-false(job.target.free) then
               // thread now owns job resources
               if job.state = READY then
                   job.state ← PROCESSING;
                   Sequential BFS-DD expansion of states in the frontier pertaining to
                   job.source with duplicate detection against the partition of Open ∪ Close
                   corresponding to job.target;
                   If a goal is encountered, reconstruct the i-plan and write it to solution;
                   job.state ← FINISHED;
               end
               job.target.free ← true;
           end
       end
   end
9  // Clean-up by calling context after threads terminate
   return solution;
```

ability (line 5). A second barrier (line 6) ensures that parallel layer expansion only begins after initialization is finished. Each thread scans the job-array for waiting jobs. Note that there is no synchronization on this structure. It then attempts to attain ownership of the job's target resource by executing an atomic test-and-set on the corresponding binary variable. Upon success, the job state has to be tested again as "between" lines 7 and 8, a cooperating thread might have acquired the job, processed it and released the resource. If that test passes, the job is executed by generating successors for the job's operators of all states in the frontier corresponding to the the job's source partition id. The edge partitioning guarantees consistency during this partial expansion as figure 5.7 shows.

If a goal state is encountered during this expansion, the corresponding i-plan is written to the synchronized $solution$ variable and a termination request send to all worker threads. After processing the job, the thread updates the job state accordingly and releases the resource. After the worker threads terminate, the calling context simply returns the contents of the $solution$ variable. Note that with exception of accesses to $solution$ and the synchronization by barriers at the beginning of each layer, the algorithm does not make use of any platform level synchronization primitives with their associated costly system calls. This is particularly important when using relatively fine-grained abstractions that result in a comparatively large number of individually small jobs.

For any parallel algorithm, it is important to prove the absence of deadlocks. Fortunately, the proof is trivially simple for parallel edge partitioning, because it breaks one of the four necessary conditions for a deadlock. In computer science, it is common knowledge that these conditions are (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait, and that breaking any one of the four is sufficient to prevent a deadlock from happening. The condition that is never satisfied in parallel edge partitioning is "hold and wait," because each task only requires a *single* shared resource. As parallel edge partitioning is deadlock-free by design, it has no overhead for deadlock detection or avoidance. For example, the SDD-based PBNF employs machinery to detect and resolve such deadlocks, which in turn can lead to lifelocks which to avoid required even more machinery [BLZR09] - in the end, its authors had to fall back to complex verification tools to guarantee its functioning.

In contrast, parallelization based on PEP is very simple on a conceptual level and that simplicity transfers into implementation practice - to great benefit as developing and debugging parallel algorithms remains challenging. Listing 22 serves as an example that PEP based algorithms stay manageably simple, even when avoiding the use of operating system synchronization primitives in favor of faster but more complicated, user-space based synchronization.

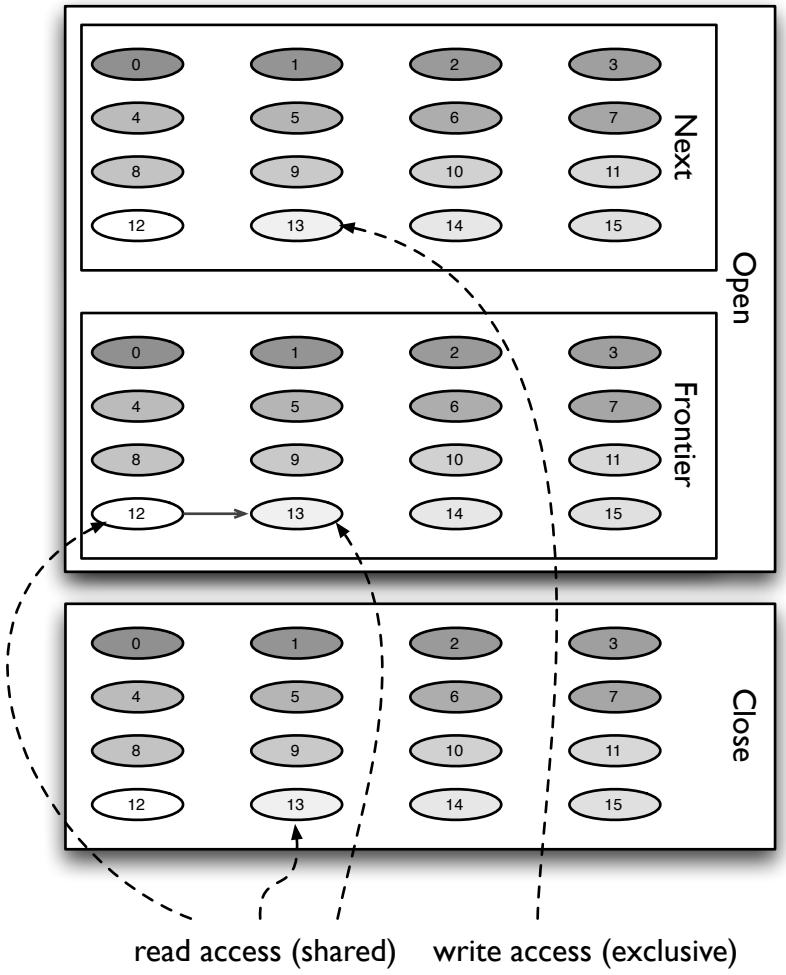


FIGURE 5.7 Expansion of partition 12 by a thread along edge $(12, 13)$ for a 15-puzzle instance partitioned according to position of the blank tile. The thread reads partition 12 of the frontier for the original states and partitions 13 from $Close$, the frontier and next for duplicate detection. If states pass the test, they are written to partition 13 in next. PEP guarantees that no other thread reads or writes partition 13 of next concurrently. Note that other threads can (and do) concurrently *read* partition 13 in the frontier.

5.3 Empirical Comparison of PEP, PSDD, PBNF and HDA*

Now that I described the technique and gave an example of its integration, it is time to revisit the original motivation for its development. That is, to expand the heuristic search toolkit with a conceptually simple but effective parallelization model. In the following I will compare the performance PEP-based BFHS (EP), PSDD-based BHFS (SDD), PBNF and HDA* on a variety of IPC instances. Breadth-first heuristic search [ZH04a] differs from parallel BFS-DD in that it (1) only retains the "hull" of *Close* but still guarantees full duplicate detection with the smaller memory footprint, (2) recreates an i-plan not directly, but instead keeps an intermediate layer that upon discovery of a goal state allows it to split the problem into two (much simpler) subproblems to which it recursively applies itself (a significantly more clever version of what was sketched in section 4.2.13) and (3) uses a heuristic to prune states if their admissible estimate exceeds a pre-supplied upper-bound. In concept and effect (apart from the pruning) it is a breadth-first search with a low memory footprint at the cost of some computational overhead and a significant increase in algorithmic complexity. I refer interested readers to [ZH04a, ZH06] for a much more in-depth treatment. The implementations of PEP, PSDD and HDA* used in this evaluation were programmed by my colleague Rong Zhou. The test uses relatively large instances from different IPC domains. The experiments were performed on dual quad-core Intel Xeon workstation. For a short introduction of these domains, see section 4.2.13. To put the following results into context, it is my understanding that (at the time of writing) single-threaded (BFHS-)SDD is amongst the fastest sequential algorithms available.

5.3.1 Successor Generation

I want to start this comparison by a quick look at successor generation. The abstract state graph offers a computationally trivial method of speeding up this process. Consider figure 5.8. The usual pSTRIPS encoding of the 8-puzzle sports 192 operators in total, each corresponding to one particular configuration of the blank and a neighboring tile. For the above abstraction to the position of the blank tile, each abstract transition corresponds to 8 of these operators. In general, the successor generator tests the preconditions of each operator and if they apply creates the corresponding successor. Using the partitioning of *Open* and the abstract graph one can immediately discard the vast majority of these operators as they are bound to fail their precondition tests. In the 8-puzzle example, this leaves between 16 (for states in partitions $0, 2, 6, 8$) to 32 (for states in partition 4) potentially applicable operators. For larger problems, this ratio only widens (e.g. 720 operators versus at most 60 for the 15-puzzle). A further benefit is that the operator descriptions themselves can be simplified. In the example, the

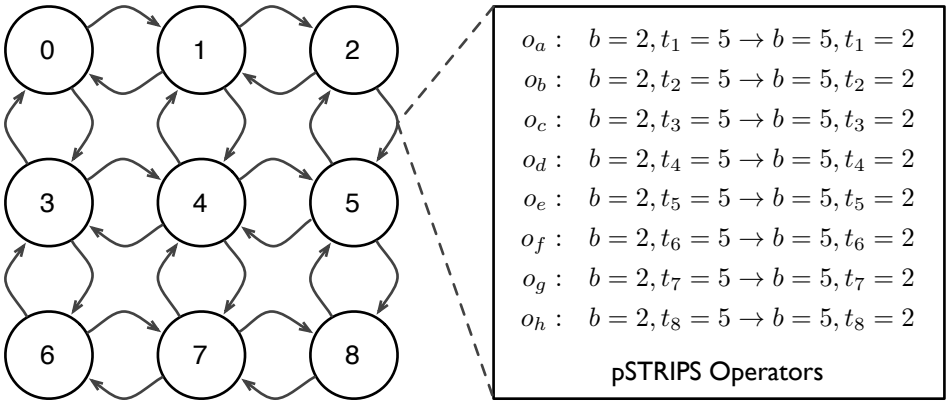


FIGURE 5.8 Abstract state graph on the position of the blank in the 8-puzzle with corresponding pSTRIPS operators for the transition from 2 to 5.

atomic precondition that blank must be in position 2 is trivially fulfilled for all states in that partition and hence needs not to be evaluated. Exploiting this property can dramatically reduce the amount of computation necessary for successor generation.

To this end, see table 5.1 for a comparison of the number of states generated per second when running FD, SDD and EP with no parallelization and a blind heuristic. FD is known for its state of the art successor generator using efficient, decision tree based precondition checking (see [Hel06a] for details). The staggered expansion of EP with the ex aequo smaller duplicate detection scopes furthermore leads to much better spatial locality of memory accesses and hence less cache misses. Unsurprisingly the extend of this benefit depends on domain structure and chosen abstraction as the table shows. On average SDD is about $3.2\times$ and EP about $5.8\times$ faster than FD.

5.3.2 Performance and Scaling

In heuristic STRIPS planning however, it is fairly usual that evaluating a state takes longer than generating it. Thus the comparison between EP, SDD and FD is with their default admissible heuristics turned on. While the heuristics used are not the same, they are all based on abstractions: the merge&shrink heuristic [HHH07] for FD and pattern database heuristics for both SDD and EP.

SDD and EP were limited to 2 GB of RAM, FD was allowed the machine's full 4 GB as there is no simple way to limit its peak memory usage. For the merge&shrink heuristic, an abstraction size of 1000 was set, identical to what was used by [KFB09]. The runtime results are given in table 5.2 for SDD and table 5.3 for EP. These problems are the largest in each

	FD	SDD		EP	
problem	St/sec	St/sec	× FD	St/sec	× FD
driverlog-14	1.1M	1.4M	123%	1.5M	138%
depots-13	0.63M	1,6M	263%	2.0M	319%
logistics-9	0.67M	3.2M	476%	3.7M	538%
freecell-5	0.30M	0.26M	87%	1.6M	545%
blocks-10	0.41M	2.0M	475%	2.4M	572%
satellite-5	0.76M	2.6M	328%	6.3M	786%
gripper-8	0.53M	2.6M	494%	6.2M	1,175%

TABLE 5.1 Brute-force speed (million of states per second) comparison of FD, SDD, and EP on pSTRIPS planning problems.

Problem	Len	Exp	1 Thrd	2 Thrd	7 Thrd	$\lvert\tilde{G}\rvert$
logistics-9	36	2,032,316	4.84	3.26	2.05	14,641
depots-13	25	1,924,439	16.50	10.79	7.71	625
satellite-5	15	3,162,393	28.68	23.14	20.71	1,331
elevator-12	40	24,223,337	84.00	47.13	22.81	1,600
gripper-8	53	66,906,969	95.30	64.10	48.59	3,600
blocks-10	34	119,755,718	202.44	129.56	77.23	2,197
freecell-5	30	16,633,205	423.10	294.86	242.85	2,304
driverlog-14	28	46,243,536	446.28	234.25	126.44	784

TABLE 5.2 Results for SDD on IPC planning problems with 1, 2 and 7 threads. Columns show solution length (Len), number of node expansions (Exp), runtime in wall-clock seconds, and the size of the abstract graph ($|\tilde{G}|$).

Problem	Len	Exp	1 Thrd	2 Thrd	7 Thrd	$\lvert\tilde{G}\rvert$
logistics-9	36	10,623,253	4.63	3.07	1.96	1,331
depots-13	25	28,027,766	14.94	8.15	3.66	625
satellite-5	15	60,083,105	18.61	11.17	10.87	121
elevator-12	40	76,038,474	58.69	32.28	12.46	64
gripper-8	53	348,411,069	50.88	46.39	39.97	180
blocks-10	34	949,979,726	215.47	154.65	132.29	169
freecell-5	30	65,609,725	75.34	48.76	36.57	64
driverlog-14	28	664,970,840	353.13	191.49	84.98	784

TABLE 5.3 Results for EP on IPC planning problems with 1, 2 and 7 threads. Columns show solution length (Len), number of node expansions (Exp), running time in wall-clock seconds, and the size of the abstract graph ($|\tilde{G}|$). Note that EP expansions are staggered, hence the higher numbers.

of the eight domains that can be solved within 2GB of RAM. While EP requires $10\times$ more staggered node expansions than SDD needs full node expansions, it is still significantly faster. This reflects PEP's faster precondition checking. It also shows that PEP can be equally or more effective with a coarser abstraction function than SDD. Note that for this comparison, EP, on average, uses abstract graphs that are more than $10\times$ smaller than SDD's. This not only conserves memory, it is also faster to traverse and manipulate a smaller abstract graph. However, EP can leverage sufficient concurrency out of an abstract graph with as few as 64 abstract nodes. Note that if SDD (or PBNF for that matter) used an abstract graph that small, its wall-clock runtime would be significantly worse than what is presented in Table 5.2.

FD ran on the same instances but only managed to solve satellite-5 and freecell-5 and ran out of memory on the rest of the problems except for elevator-12 (not supported by FD), hence no runtime is given. While all planners become slower with heuristics turned on, their relative speed *remains* roughly the same (while the heuristics are different, they are both lookup based). Measuring the number of states *generated and evaluated* per second, on average, EP is $1.85\times$ faster than SDD, which in turn is $3.05\times$ faster than FD. FD was also run on a machine with 96 GB of memory, which enabled it to solve 3 more problems. Even here, logistics-9 and driverlog-14 remain unsolvable.

Understanding the relative speed difference of sequential algorithms is important when comparing their parallelized counterparts. For example, the most commonly used metric for

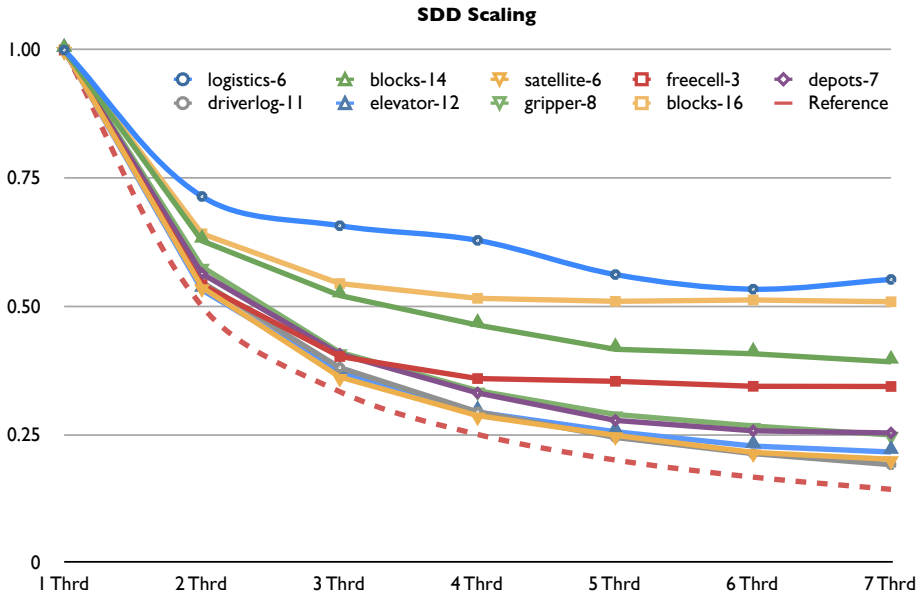


FIGURE 5.9 Relative runtime of SDD as a function of participating threads on different IPC problems. Reference is an ideal, perfectly parallelizable workload.

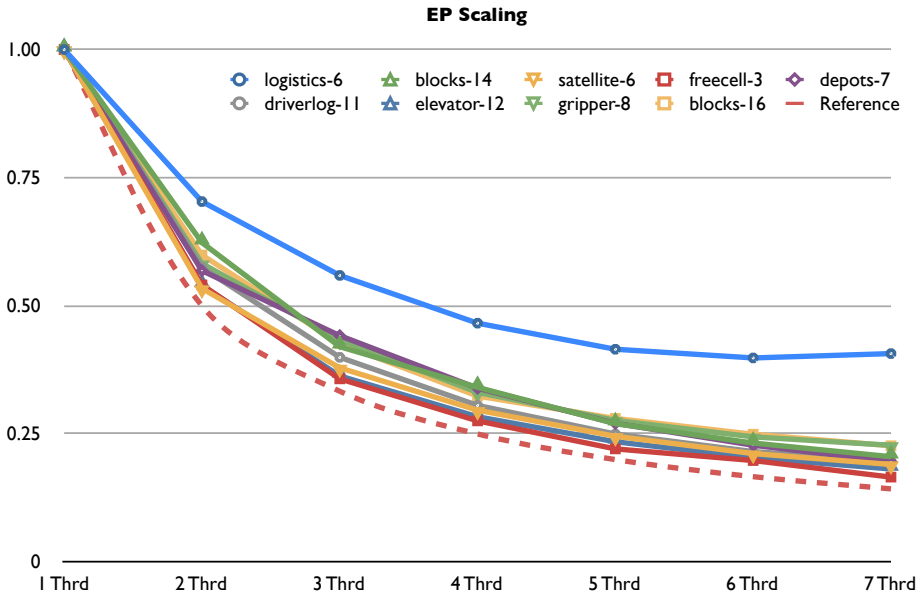


FIGURE 5.10 Relative runtime of EP as a function of participating threads on different IPC problems. Reference is an ideal, perfectly parallelizable workload.

	HDA*		PBNF	
Problem	2 Thrd	7 Thrd	2 Thrd	7 Thrd
logistics-9	6.67	3.39	4.14	2.28
depots-13	15.48	6.27	10.13	9.20
satellite-5	28.86	11.05	59.20	MEM
elevator-12	96.58	50.66	31.34	12.85
gripper-8	178.83	58.26	89.79	67.97
blocks-10	MEM	MEM	MEM	MEM
freecell-5	MEM	MEM	183.75	226.90
driverlog-14	MEM	MEM	MEM	MEM

TABLE 5.4 Runtime (in seconds) of HDA* and PBNF on IPC planning problems with 2 and 7 threads. MEM denotes instances where the algorithm failed as it ran out of memory.

parallel efficiency is the so-called speedup ratio, which measures the performance of a parallel algorithm against its corresponding sequential implementation rather than its direct competitors. This can be misleading as the same search algorithm with a slow successor generation function achieves ex aequo higher parallel speedups than with a fast one. Here for example, the speed difference in successor generation between EP and FD is over a factor of 5.6, which means the threads of EP must synchronize 5.6 times *less frequently* with one another just to achieve the same *relative* speedup ratios as FD, if both use synchronization mechanisms with comparable overhead (e.g., POSIX threads). Graphs 5.9 and 5.10 give the parallel speedups for SDD and EP (now with equal abstractions). Under these conditions, with exception of the logistics instance, EP shows almost perfect scaling.

The original HDA* is built on top of FD, which as table 5.1 shows has a slower successor generator than the other planners used. To ensure a fair comparison, HDA* was re-implemented to make sure that it is comparable to SDD in speed and uses exactly the same admissible pattern database heuristic.

Table 5.4 shows the runtime of HDA* and PBNF (the lifelock-free version) on the same set of planning problems. The PBNF implementation used was the one described in [BLZR09]. All algorithms used the same admissible heuristic (computed by the same C code).

Before going into the details, a few important remarks. HDA* and PBNF represent the current state of the art in parallel heuristic so they make for obvious comparison candidates.

Nevertheless, it is somewhat problematic to compare breadth-first heuristic search with a best-first search algorithm like A* or its parallel variants, such as HDA* and PBNF, simply on the grounds of parallel-speedup ratios or runtime. Despite its significant computational complexity, memory is usually the limiting factor in pSTRIPS planning (c.f. chapters 3 and 4) so trading spatial for temporal efficiency is seldom viable in practice. Hence in addition to the general reservations with the parallel-speedup metric stated above, I want to elaborate on two specific problems arising from this comparison.

Problem #1 The underlying rationale for using a breadth-first instead of a best-first approach to heuristic search is that the former induces a smaller search frontier than the latter. Moreover, this difference increases as more accurate heuristics are employed [ZH06]. Since the frontier can be seen as a snapshot of the "workload" of the search algorithm, A* has the inherent advantage of having a larger available work-pool (from which to create parallel work tasks) than breadth-first heuristic search at any point in time.

Reason #2 The parallel version of breadth-first heuristic search uses (g-cost) layer-based synchronization to conserve memory. As such there is a higher chance of starvation as the algorithm approaches the end of a layer expansion. PBNF is not layer-based. The implementation of HDA* is f-estimate layer-based (i.e., states with equal f-estimates form a layer) and there are always fewer f-estimate layers than there are g-cost layers. For example with a perfect heuristic h^* would, f-estimate layer-based best-first search would only need to expand nodes from a single layer (i.e. basically remove the need of layer-based synchronization for HDA*).

Despite all these disadvantages, the results shows that EP runtime compares very favorably with SDD, HDA* and PBNF over all thread counts. The only exception is blocks-10 for which EP needed more incremental expansions than the number of nodes generated and was outperformed by SDD. (i.e. on average states generated less than one successor each during staggered expansions). These results are remarkable, as HDA* and PBNF are considerably less memory-efficient than BFHS which prevents them from solving larger problems. One interesting observation is that PBNF ran out of memory on satellite-5 with 7 threads, but not with 2 threads. Unlike PEP, PBNF's memory requirements can increase with the number of threads used. For satellite-5, PBNF's peak number of states held in memory are roughly 16, 32, and 54 million for 1, 2, and 4 threads.

Another interesting question raised by PBNF's results on satellite-5 is whether it is still appropriate to view it as a best-first search, especially when it is run on a (relatively) large

Problem	1 Thrd	2 Thrd	3 Thrd	4 Thrd	5 Thrd	6 Thrd	7 Thrd
15-puzzle-17	1,488.40	810.14	565.81	454.22	384.15	352.26	315.14
15-puzzle-53	1,111.11	594.92	411.61	317.96	268.30	235.38	213.79
15-puzzle-56	722.65	398.54	280.22	219.57	193.36	174.45	156.92
15-puzzle-59	1,001.89	561.94	407.03	326.94	285.92	251.35	240.54
15-puzzle-92	964.70	521.49	365.13	282.13	238.52	209.64	191.97

TABLE 5.5 Runtime of SDD (in seconds) on a selection of Korf's 100 instances.

Problem	1 Thrd	2 Thrd	3 Thrd	4 Thrd	5 Thrd	6 Thrd	7 Thrd
15-puzzle-17	425.13	294.43	243.15	219.50	203.93	196.14	190.62
15-puzzle-53	333.47	238.06	195.12	176.59	163.20	156.98	154.04
15-puzzle-56	246.44	183.16	150.73	137.82	129.29	125.09	121.99
15-puzzle-59	309.83	212.66	181.96	164.83	153.28	145.59	141.82
15-puzzle-92	270.65	192.65	161.73	146.97	136.61	133.16	128.76

TABLE 5.6 Runtime of EP (in seconds) on a selection of Korf's 100 instances.

number of threads. While the safe version of PBNF (using a technique called "hot $n$blocks" where an $n$block is essentially a partition) guarantees a livelock-free search, its node expansion order can deviate arbitrarily from best-first order. In the extreme, if a partition on $Open$ is not interfering with a better one and is the only one the free list, PBNF will choose it for expansion *no matter* how bad (from a best-first perspective) that partition is, resulting in what is basically a random expansion order.

Finally, SDD and EP were run on Korf's 100 15-Puzzle instances encoded as STRIPS planning problems. Tables 5.5 and 5.6 give their respective runtimes for a selection of instances. Note that the previous best planner can solve 93 of them in hours [HBH+07]. With 2 GB of RAM, EP solves 95 of them in minutes, using a *weaker* admissible heuristic. Note that none of Korf's 100 instances can be solved by FD or the original HDA* - both ran out of memory on the 96 GB machine when attempting to solve the easiest instance (#79). On average, EP achieves 3× speedup in precondition checking over SDD in this domain (c.f. figure 5.8).

5.4 Summary

In this chapter, I described parallel edge partitioning and showed how it allows to exploit structural domain properties in a principled way in order to efficiently parallelize graph search algorithms. Parallel edge partitioning has several theoretical and practical advantages over its related approach, parallel structured duplicate detection. By establishing a conception-ally simple, deadlock free synchronization regime, it significantly reduces the complexity of parallel search algorithm design. By reducing the size of duplicate detection scopes, it increases the degree of synchronization-free concurrency and improves memory system performance through higher spatial locality of loads and stores. In the context of domain-independent propositional planning, edge partitioning allows for an expansion regime that significantly speeds up precondition checking during successor generation making it a worthwhile technique even for sequential algorithms.

CHAPTER 6

Conclusion

In this thesis, I have introduced two techniques to widen the memory and I/O bottleneck in explicit state planning on the one side and enable the exploitation of ever more parallel hardware on the other. Due to algorithmic advances over the recent years, spatial complexity has established itself more and more as the biggest bottleneck in state-of-the-art propositional planners. As a result, space-efficient representation has more and more moved into the center of attention of the academic community. Binary Decision Diagrams quickly emerged as the technology of choice even though they have proven to be difficult citizens. Their benefits are limited to a subset of domains where they often only work well when few, large sets need to be represented. Their integration is challenging as they offer no easy (space efficient) way to associate the ancillary data to elements most search algorithms rely on. Their computational overhead for explicit state search is significant. Despite all these shortcomings, over a wide range of problem domains, a planner based on BDDs outperformed all other state-of the art planners at the last international planning competition [HDR08].

The development of the Level Ordered Edge Sequence encoding technique was strongly motivated by these results. LOES encodings result in state-set representations that are consistently small and computationally efficient for sets of all sizes. LOES spans an address space over its constituents which enables time and space efficient association of additional data to its constituents and hence integrates naturally into a wide range of state-of-the-art algorithms and memoization techniques in the context of combinatorial search and optimization. Furthermore, its close conceptional relationship with binary decision diagrams allows for straightforward interoperability between the two representations. Such hybrid representations, where LOES serves as a fallback for domains which are not amendable to BDD compression, their computational overhead is deemed too high or the resulting limitation in applicable search algorithms is impractical, allow to alleviate the memory bottleneck of domain-independent propositional planners by orders of magnitude.

In recent years, parallel computing has become the norm rather than the exception in nearly all deployed hardware platforms, from high-end servers to mobile phones. While the depth-first family of search-algorithms adapts more or less straightforwardly to this form of computation, breadth and best-first algorithms are much more challenging to parallelize. Still, in most domains the latter, with their ability to exploit overlapping subproblems and optimal substructure, outperform the former (even when parallelized) by orders of magnitude. This ability relies on the runtime accumulation of vast amount of state and its manipulation at a high frequency in a *consistent* way. That need for consistency results in so much overhead due to communication and synchronization that straightforward parallelizations based on transactional data structures generally give worse performance than their sequential counterparts (see [BLZR09] for some empirical evidence). Efficient parallel breadth and best-first search necessitates changes on the algorithmic level, but even with current tools, designing and debugging explicitly parallel algorithms remains challenging. This may be the reason for the surprisingly little work in this field so far. On thread of work has focused on explicitly distributing search state, thereby mostly eliminating the need for synchronization and focused to deploy on very high end supercomputers to manage the ensuing communications load (i.e.[EHMN95] and [KFB09]). Performance and scaling of these approaches on of-the-shelf hardware is predictably lacking.

The other thread of work based on structured duplicate detection has focused on domain abstraction to exploit domain inherent concurrency allowing to lift synchronization to the level of a few, large partitions of shared state with drastically reduced frequency. However, SDD depends on special domain properties and is hence not generally applicable. The technique also necessitates careful resource handling to avoid dead- and lifelocks, markedly complicating algorithm design. Last but not least, the one relatively complex parallel best-first algorithm hitherto developed on top of these ideas (i.e. PNBF [BLZR09]) trades off spatial-efficiency for its scalability in a substantial way, an at best questionable design choice given the realities of propositional planning. Parallel Edge Partitioning more or less removes these barriers of entry. It represents a conceptually simple and powerful synchronization scheme that exploits basic domain structure for very good concurrency, regardless of any inherent properties. Most importantly, it avoids most of the complexities of parallel algorithm design by being inherently deadlock free. In fact, it enabled developing and debugging a parallel version of breadth-first heuristic search, one of the fastest (but also relatively complex) explicit state search algorithms in less than a week. The resulting algorithm shows very good speed increases over its sequential base and consistently outperformed the other (publicized) parallel breadth- and best-first approaches while retaining BFHS' frugal memory requirements. In fact it produced

the best results on Korf's 100 instances (of the 15-puzzle) hitherto published for any domain independent planner.

In summary, both are widely applicable technologies that fill important gaps in the map of heuristic search techniques. As such they should transfer naturally to the field of domain dependent planning. A particularly interesting domain during my work at PARC were target-value path problems. They represent a challenging, novel class of combinatorial optimization problems with practical applications. The development of depth-first target value search shows how a toolkit of well defined and understood heuristic search techniques can be used to integrate domain knowledge and economically derive efficient domain specific solvers. The work on target value search serves as a good example of the cross-fertilization between domain-dependent and domain-independent planning.

CHAPTER 7

Outlook

The bulk of research presented in this thesis was driven by our groups' "Planning in the Cloud" vision. The overarching idea in a single sentence is to provide a domain independent planner with a sufficiently expressive problem description language that can be deployed in commercial cloud computing environment and scales to large problem sizes. Large combinatorial optimization and search problems permeate our environment. The potential spoils are often sizable. A primary example is the ever more elaborate logistics and production planning which enables many large organizations to run more efficient "just-in-time", "lean-manufacturing" or "agile-manufacturing" regimes than their smaller competitors. Computing (near-) optimal solutions to such problems has thus far often been the privilege of either large organizations or extensive collaboration as their set-up cost is often sizeable, partly due to the necessary elaborate algorithm development and implementation effort and partly due to the often massive computational power required for feasibly running the resulting solvers.

The on-demand availability of large parallel clusters at low hourly rates in the form of cloud computing services together with high-performance, general combinatorial optimizers could on the one hand make these powerful tools available to a much wider audience and on the other hand make optimization techniques economically feasible for a wide range of problems.

The techniques described here represent the very first step towards that aim. Parallel Edge Partitioning provides a natural framework for a principled mapping of shared search state across multiple distributed collaborating machines each sporting multiple processing elements. The primary pain point for high performance computing in current commercial cloud environments is the excessive latency and lacking bandwidth of inter-node links (see [EH08], [HH09] and [RVG+10] amongst many others). Edge partitioning was originally devised as a technique to ease the communications load in external memory search and hence expands straightforwardly from an intra-node (shared-memory) synchronization technique to the inter-node distributed memory case helping to lower the frequency and scope of necessary communications.

Compression techniques such as LOES can act synergistically in two ways - first partitioning the search state according to structural domain properties through parallel edge partitioning allows LOES to even better exploit the memory at each node (i.e. all states at a pep-node share assignments to subsets of their state variables leading to ceteris paribus longer common prefixes in the partitions) and second succinctly encoding state-sets that have to be moved between nodes can significantly ease the bandwidth bottleneck.

Such large scale heuristic search problems also represent a natural ecosystem for memoization based heuristics. Pattern databases and merge & shrink heuristics have the inherent advantage that estimates are reused many times during a search. Particularly for large problems, this surmounts to a sizable reduction in computational complexity to heuristics that require costly re-computation for every state generated. One of the biggest limitations to their wide-spread use in domain independent planning has been that the spatial efficiency of the common perfect hash representation degenerates quickly as the underlying abstractions became more fine grained (i.e. for stronger heuristics). Here the especially the combination of LOES and BDDs looks extremely promising, robustly offering up to several order of magnitude better space efficiency for large PDBs. In all these advances open up several concrete, interesting research questions for the future such as automatic pattern selection in domain independent planning for strong PDB heuristics, memoization techniques for h^m and landmark style heuristics (i.e. the two other important families of heuristics [HD09]) to make them computationally feasible when large amounts of states need to be traversed, optimal mappings of a problem's abstract graph's nodes to the platform's computing topology such that communications between distributed memory environments are minimized and adaptive load-balancing schemes for such massively parallel search algorithms.

Bibliography

[ACNS10] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. *Proc. 11th ALENEX*, pages 84–97, 2010.

[Bae95] C. Baeckstroem. Expressive equivalence of planning formalisms. *Artificial intelligence*, 76(1-2):17–34, 1995.

[Bay72] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[BB08] D. Bryce and O. Buffet. 6th international planning competition: Uncertainty part. In *Proceedings of IPC*. Citeseer, 2008.

[Bel78] R. Bellman. *An introduction to artificial intelligence: can computers think?* Boyd & Fraser Pub. Co., 1978.

[BF97] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[BG01] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BH08] M. Ball and R.C. Holte. The compression power of symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–11, 2008.

[Bja08] P. Bjarnolf. Threat analysis using goal-oriented action planning. Master's thesis, University of Skövde, 2008.

[BK91] C. Backstrom and I. Klein. Parallel non-binary planning in polynomial time. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 268–273. Citeseer, 1991.

[BK10] Teresa Maria Breyer and Richard E. Korf. 1.6-bit pattern databases. In *National Conference on Artificial Intelligence*, 2010.

[BLRZ09] E. Burns, S. Lemons, W. Ruml, and R. Zhou. Suboptimal and anytime heuristic search on multi-core machines. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS-09)*, 2009.

[Blu95] A.L. Blum. Fast planning through planning graph analysis. Technical report, DTIC Document, 1995.

[BLZR09] E. Burns, S. Lemons, R. Zhou, and W. Ruml. Best-first heuristic search for multi-core machines. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 449–455. Morgan Kaufmann Publishers Inc., 2009.

[BN95] C. Backstrom and B. Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[BW02] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *Computers, IEEE Transactions on*, 45(9):993–1002, 2002.

[Byl91] T. Bylander. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, volume 1, pages 274–279. Citeseer, 1991.

[Byl94] T. Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[CGGT97] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for ar. *Recent Advances in AI planning*, pages 130–142, 1997.

[Cha85] E. Charniak. *Introduction to artificial intelligence*. Pearson Education India, 1985.

[CKT91] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th IJCAI*, pages 331–337. Citeseer, 1991.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to algorithms. *The Massachusetts Institute of Technology. New York*, 1990.

[CS96] J. Culberson and J. Schaeffer. Searching with pattern databases. *Advances in Artifical Intelligence*, pages 402–416, 1996.

[CS98] J.C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Dar70] Brad Darrach. Meet shakey, the first electronic person. *Life Magazine*, 69(21):58–68, 1970.

[DEZGK11] Carmel Domshlak, Ziv Even-Zur, Yannai Golany, and Erez Karpas. Command and control training centers: Computer generated forces meet classical planning. In *System Demo Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS-2011)*, 2011.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DK07] P.A. Dow and R.E. Korf. Best-first search for treewidth. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1146. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[DP85] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *Journal of the ACM (JACM)*, 32(3):505–536, 1985.

[DPV06] S. Dasgupta, C.H. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, Inc. New York, NY, USA, 2006.

[Dre02] S. Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, pages 48–51, 2002.

[DRR06] O.N. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. *Experimental Algorithms*, pages 134–145, 2006.

[dS58] A. de Segner. Enumeratio modorum, quibus figurae planae rectilineae per diagonales dividuntur in triangula. *Novi Commentarii Acad. Sci. Petropolitanae*, 7:203–209, 1758.

[DS07] K. Doris and D. Silvia. Improved missile route planning and targeting using game-based computational intelligence. In *Computational Intelligence in Security and Defense Applications, 2007. CISDA 2007. IEEE Symposium on*, pages 63–68. IEEE, 2007.

[DW91] J. Doyle and M.P. Wellman. Impediments to universal preference-based default theories. *Artificial Intelligence*, 49(1-3):97–128, 1991.

[Ede02] S. Edelkamp. Symbolic pattern databases in heuristic search planning. *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 274–283, 2002.

[EH00] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. *Recent Advances in AI Planning*, pages 135–147, 2000.

[EH01] S. Edelkamp and M. Helmert. Mips: The model-checking integrated planning system. *AI magazine*, 22(3):67, 2001.

[EH08] C. Evangelinos and C.N. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2. *ratio*, 2(2.40):2–34, 2008.

[EHMN95] M. Evett, J. Hendler, A. Mahanti, and D. Nau. Pra*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25(2):133–143, 1995.

[EK08a] S. Edelkamp and P. Kissmann. Gamer: Bridging planning and general game playing with symbolic search. *Sixth International Planning Competition Booklet (ICAPS 2008)*, 143, 2008.

[EK08b] S. Edelkamp and P. Kissmann. Limits and possibilities of bdds in state space search. *KI 2008: Advances in Artificial Intelligence*, pages 46–53, 2008.

[Eli74] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[ENS92] K. Erol, D.S. Nau, and VS Subrahmanian. On the complexity of domain-independent planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 381–381. Citeseer, 1992.

[EQ01] E. El-Qawasmeh. Beating The Popcount. *International Journal of Information Technology*, 9(1):1–18, 2001.

[ER98] S. Edelkamp and F. Reffel. Obdds in heuristic search. *KI-98: Advances in Artificial Intelligence*, pages 81–92, 1998.

[FKH04] A. Felner, R.E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22(1):279–318, 2004.

[FL03] M. Fox and D. Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.

[FN71] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.

[Fro02] M.P.J. Fromherz. Planning and scheduling reconfigurable systems with regular and diagnostic jobs, October 30 2002. US Patent App. 10/284,560.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GN91] N. Gupta and D.S. Nau. Complexity results for blocks-world planning. In *Proceedings of AAAI-91*, volume 629. Citeseer, 1991.

[Gor65] E.M. Gordon. Cramming more components onto integrated circuits. *Electronics Magazine*, 4, 1965.

[Got97] L.S. Gottfredson. Foreword to" intelligence and social policy.". *Intelligence*, 24(1):1–12, 1997.

[Hau89] J. Haugeland. *Artificial intelligence: the very idea*. The MIT Press, 1989.

[HBG05] P. Haslum, B. Bonet, and H. Geffner. New admissible heuristics for domain-independent planning. In *Proceedings of the National Conference on Artificial*

Intelligence, volume 20, page 1163. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[HBH+07] P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1007. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[HD09] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: WhatÂŠs the difference anyway. In *Proc. ICAPS*, volume 9, pages 162–169, 2009.

[HDR08] M. Helmert, M. Do, and I. Refanidis. Results of the international planning competition 2008. *http://ipc.informatik.uni-freiburg.de/*, 2008.

[Hel03] M. Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[Hel06a] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[Hel06b] M. Helmert. New complexity results for classical planning benchmarks. In *Proceedings of the sixteenth international conference on automated planning and scheduling (ICAPS 2006)*, pages 52–61, 2006.

[Hel10] M. Helmert. Landmark heuristics for the pancake problem. In *Third Annual Symposium on Combinatorial Search*, 2010.

[HG00] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proc. AIPS*, pages 140–149. Citeseer, 2000.

[HH99] R.C. Holte and I.T. Hernádvölgyi. A space-time tradeoff for memory-based heuristics. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 704–709. JOHN WILEY & SONS LTD, 1999.

[HH09] Z. Hill and M. Humphrey. A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster? In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 26–33. IEEE, 2009.

[HHH07] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*, volume 2007, pages 176–183, 2007.

[HL10] M. Helmert and H. Lasinger. The scanalyzer domain: Greenhouse logistics as a planning problem. *Proc. ICAPS 2010*, pages 234–237, 2010.

[HLH97] B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51, 1997.

[HN01] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302, 2001.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[HNR72] P.E. Hart, N.J. Nilsson, and B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Bulletin*, (37):28–29, 1972.

[Hol97] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[Hol10] R.C. Holte. Common misconceptions concerning heuristic search. In *Third Annual Symposium on Combinatorial Search*, 2010.

[HR10] Malte Helmert and Gabriele Röger. Relative-order abstractions for the pancake problem. In *European Conference on Artificial Intelligence*, pages 745–750, 2010.

[HRK] M. Helmert, G. Röger, and E. Karpas. Fast downward stone soup: A baseline for building planner portfolios. In *PAL 2011 3rd Workshop on Planning and Learning*, page 28.

[Hun97] N.N.W. Hung. *Exploiting symmetry for formal verification*. University of Texas at Austin, 1997.

[ISY02] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pages 472–475. IEEE, 2002.

[Jac88] G.J. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, USA, 1988.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

[Jar60] M.P. Jarnagin. Automatic machine methods of testing pert networks for consistency. Technical report, K-24/60, US Naval Weapons Lab., Dahlgren, Va, 1960.

[JBV02] Rune M. Jensen, Randy E. Bryant, and Manuela M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings of AAAI-2002*, pages 668–673, Edmonton, Canada, August 2002.

[Kat87] R. Katriel. Three highly parallel computer architectures and their suitability for three representative artificial intelligence problems. 1987.

[KD09] E. Karpas and C. Domshlak. Cost-optimal planning with landmarks. In *Proc. IJCAI*, volume 9, 2009.

[KF02] R.E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134(1-2):9–22, 2002.

[KF07] R.E. Korf and A. Felner. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2324–2329, 2007.

[KFB09] A. Kishimoto, A. Fukunaga, and A. Botea. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*, 2009.

[Kle90] C.B.I. Klein. Planning in polynomial time. In *Expert systems in engineering: principles and applications: international workshop, Vienna, Austria, September 24-26, 1990, proceedings*, volume 462, page 103. Springer, 1990.

[Knu73] D.E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Addison Wesley Publishing Company, 1973.

[Koe01] J. Koehler. From theory to practice: AI planning for high performance elevator control. *KI 2001: Advances in Artificial Intelligence*, pages 459–462, 2001.

[Kor85] R.E. Korf. Depth-first iterative-deepening an optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[Kor97] R.E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the National Conference on Artificial Intelligence*, pages 700–705. JOHN WILEY & SONS LTD, 1997.

[Kor04] Richard E. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence*, pages 650–657, 2004.

[KPD+10] L. Kuhn, B. Price, M. Do, J. Liu, R. Zhou, T. Schmidt, and J. de Kleer. Pervasive diagnosis. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 40 Issue 5(10):932–944, Sept. 2010.

[KPdK+08] L. Kuhn, B. Price, J. de Kleer, M. Do, and R. Zhou. Pervasive diagnosis: Integration of active diagnosis into production plans. In *proceedings of AAAI*, 2008.

[KPSM90] R. Kurzweil, R. Productions, M.L. Schneider, and AIMS Media. *The age of intelligent machines*, volume 579. MIT press, 1990.

[KR87] V. Kumar and V.N. Rao. Parallel depth first search. part ii. analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[Kra86] E.F. Krause. *Taxicab geometry: An adventure in non-Euclidean geometry*. Dover Pubns, 1986.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence ECAI*, volume 54, pages 359–363. John Wiley & Sons, Inc., 1992.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[KS99] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 16, pages 318–325. Citeseer, 1999.

[KSH06] H. Kautz, B. Selman, and J. Hoffmann. Satplan: Planning as satisfiability. In *5th International Planning Competition*, 2006.

[KSP+08] L. Kuhn, T. Schmidt, B. Price, J. de Kleer, M. Do, and R. Zhou. Heuristic search for target-value path problem. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, 2008.

[KZTH05] R.E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM (JACM)*, 52(5):715–748, 2005.

[LB87] H.J. Levesque and R.J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational intelligence*, 3(1):78–93, 1987.

[LdKK+08] J. Liu, J. de Kleer, L. Kuhn, B. Price, R. Zhou, and S. Uckun. A Unified Information Criterion for Evaluating Probe and Test Selection. In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pages 1–8, 2008.

[Lee59] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[LN99] J. Lind-Nielsen. Buddy-a binary decision diagram package. Technical report, Technical University of Denmark, 1999.

[LNAH+01] J. Lind-Nielsen, H.R. Andersen, H. Hulgaard, G. Behrmann, K. Kristoffersen, and K.G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

[LPW79] T. Lozano-Pérez and M.A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[MC07] G. Motors and AG Continental. Tartan racing: A multi-modal approach to the darpa urban challenge. 2007.

[McC59] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, 1959.

[McD96] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 142–149, 1996.

[McM93] K.L. McMillan. *Symbolic model checking*, volume 174. Kluwer Academic, 1993.

[Mer84] L. Mero. A heuristic search algorithm with modifiable estimate. *Artificial intelligence*, 23(1):13–27, 1984.

[MGH+98] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl-the planning domain definition language. *The AIPS-98 Planning Competition Comitee*, 1998.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218(1):19–22, 1968.

[MJPL92] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[MR02] J.I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 118–126. IEEE, 2002.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 459–459. Citeseer, 1992.

[Nil98] N.J. Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann, 1998.

[NRR+68] N.J. Nilsson, C.A. Rosen, B. Raphael, L.J. Chaitin, S.E. Wahlstrom, and CA. Stanford Research Institute Menlo Park. *Application of intelligent automata to reconnaissance*. Defense Technical Information Center, 1968.

[NSS+59] A. Newell, J.C. Shaw, H.A. Simon, Rand Corporation, and International Conference on Information Processing. Report on a general problem-solving program, 1959.

[NU99] M. Nykänen and E. Ukkonen. Finding paths with the right cost. In *STACS 99*, pages 345–355. Springer, 1999.

[NU02] M. Nykänen and E. Ukkonen. The exact path length problem. *Journal of Algorithms*, 42(1):41–53, 2002.

[OBdBB+04] J. Orkin, P. Baillie-de Byl, D. Borrajo, J. Funge, M. Garagnani, P. Goetz, JJ Kelly III, I. Millington, B. Schwab, and RM Young. Working group on goal-oriented action planning. Technical report, International Game Developers Association, 2004.

[Ork03] J. Orkin. Applying goal-oriented action planning to games. *AI Game Programming Wisdom*, 2:217–229, 2003.

[Ork05] J. Orkin. Agent architecture considerations for real-time planning in games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*, 2005.

[Ork06] J. Orkin. Three states and a plan: the ai of fear. In *Game Developers Conference*, volume 2006. Citeseer, 2006.

[Pav08] Adam Pavlacka. *Interview with Kieran Bridgen, studio communications manager for Creative Assembly*, 12 2008.

[Pea84] J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.

[PMG98] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence*. Oxford University Press, 1998.

[PSP94] S. Panda, F. Somenzi, and B.F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 628–631. IEEE Computer Society Press, 1994.

[RK87] V.N. Rao and V. Kumar. Parallel depth first search. part i. implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[RK91] E. Rich and K. Knight. Introduction to artificial networks. *Mac Graw-Hill Publications, New York*, 1991.

[RNC+10] S.J. Russell, P. Norvig, J.F. Candy, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*. Prentice hall, 2010.

[RVG+10] J.J. Rehr, F.D. Vila, J.P. Gardner, L. Svec, and M. Prange. Scientific computing in the cloud. *Computing in Science & Engineering*, 12(3):34–43, 2010.

[RW90] D. Ratner and M. Warmuth. Finding a shortest solution for the n× n-extension of the 15-puzzle is intractable. *J. Symb. Comp*, 10:111–137, 1990.

[RW10] S. Richter and M. Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010.

[S+99] R.P. Stanley et al. *Enumerative Combinatorics: Volume 2*, volume 118. Cambridge university press Cambridge;, 1999.

[SA77] D.J. Slate and L.R. Atkin. Chess 4.5—the northwestern university chess program, 1977.

[Sch03] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics, vol. 24*. Springer, Berlin, 2003.

[SKP+09] T. Schmidt, L. Kuhn, B. Price, J. de Kleer, and R. Zhou. A depth-first approach to target-value search. In *Second Annual Symposium on Combinatorial Search*, 2009.

[Som97] F. Somenzi. Cudd: Colorado university decision diagram package. *Public software, Colorado Univeristy, Boulder*, 1997.

[SR86] E. Sandewall and R. Roennquist. *A representation of action structures*. Department of Computer and Information Science, Linkoeping University, 1986.

[SS06] J. Slocum and D. Sonneveld. *The 15 puzzle book*. The Socum Puzzle Foundations, 2006.

[SZ11a] Tim Schmidt and Rong Zhou. Representing pattern databases with succinct data structures. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011*, 2011.

[SZ11b] Tim Schmidt and Rong Zhou. Succinct set-encoding for state-space search. In *Proceedings of the 25th Conference on Artificial Intelligence (AAAI-11)*, 2011.

[THN04] S. Triig, J. Hoffmann, and B. Nebel. Applying automatic planning systems to airport ground-traffic control-a feasibility study. In *KI 2004: advances in artificial intelligence: 27th Annual German Conference on AI, KI 2004, Ulm, Germany, September 20-24, 2004: proceedings*, page 183. Springer-Verlag New York Inc, 2004.

[Tur50] A.M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[UE10] T. Uras and E. Erdem. Genome rearrangement: a planning approach. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21:309–315, April 1978.

[WDZF11] Ruml Wheeler, M.B. Do, R. Zhou, and M.P.J. Fromherz. On-line planning and scheduling: An application to controlling modular printers. *Journal of Artificial Intelligence Research*, 40:415–468, 2011.

[Win92] P.H. Winston. *Artificial intelligence*. Addison-Wesley, 1992.

[YdB06] B. Yue and P. de Byl. The state of the art in game ai standardisation. In *Proceedings of the 2006 international conference on Game research and development*, pages 41–46. Murdoch University, 2006.

[YL89] H. Yoo and S. Lafortune. An intelligent search method for query optimization by semijoins. *Knowledge and Data Engineering, IEEE Transactions on*, 1(2):226–237, 1989.

[YL03] H.L.S. Younes and M.L. Littman. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. In *In Proceedings of the 14th International Conference on Automated Planning and Scheduling*. Citeseer, 2003.

[ZH02] R. Zhou and E.A. Hansen. Memory-bounded a* graph search. In *Fifteenth International FLAIRS Conference (FLAIRS-02)*, 2002.

[ZH04a] R. Zhou and E. Hansen. Breadth-first heuristic search. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, pages 92–100, 2004.

[ZH04b] R. Zhou and E.A. Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 683–689. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2004.

[ZH06] R. Zhou and E.A. Hansen. A breadth-first approach to memory-efficient graph search. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1695. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[ZH07a] R. Zhou and E.A. Hansen. Edge partitioning in external-memory graph search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2410–2416, 2007.

[ZH07b] R. Zhou and E.A. Hansen. Parallel structured duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1217. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[Zob70] A.L. Zobrist. A new hashing method with application for game playing by. 1970.

[ZSH+09] Z. Zhang, N.R. Sturtevant, R. Holte, J. Schaeffer, and A. Felner. A* search with inconsistent heuristics. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, 2009.

[ZSH+10] R. Zhou, T. Schmidt, E.A. Hansen, M.B. Do, and S. Uckun. Edge partitioning in parallel structured duplicate detection. In *Third Annual Symposium on Combinatorial Search*, 2010.