Landmark-Based Heuristics and Search Control for Automated Planning

Silvia Richter
Dipl.-Inf.

A thesis submitted in fulfilment of the requirements
of the degree of Doctor of Philosophy

Institute of Intelligent and Integrated Systems
Faculty of Science, Environment, Engineering and Technology
Griffith University, Brisbane, Australia

November 2010

Abstract

A key characteristic of intelligence is the use of efficient problem-solving strategies when faced with unfamiliar tasks. Enabling machines to do autonomous problem-solving is thus a major milestone on the path to developing intelligent systems. Automated planning is a discipline in artificial intelligence research that studies this topic, specifically the process of automatically computing strategies for using actions to achieve a desired outcome. Given a declarative description of a task, a planning system finds an action sequence (a plan) that leads from a given initial state to a state that satisfies a specified goal description. The quality of a plan is measured via its length or, in cost-based planning, via associated costs of the actions it comprises. While the planning problem in general is computationally intractable, many planning tasks can be solved efficiently due to some inherent structure of the task. Knowledge about such structure or certain properties of a planning task, so-called control knowledge, can often be extracted automatically from the problem description.

This thesis makes several contributions to improve the efficiency of automated planning. We focus on forward-chaining heuristic search in the state space of a planning task, currently the most widely used approach to planning. In the first part of this thesis, we detail novel methods for extracting landmarks, a particular type of control knowledge, from planning tasks. We then propose a way of using these landmarks as a *heuristic estimator* for judging progress during planning, and show empirically that this leads to shorter plans and allows solving more tasks in unit-cost planning. We furthermore analyse the performance gain achieved via landmarks in *cost-based* planning and find that landmarks can be particularly helpful in this setting, making up for the bad performance of other (cost-sensitive) heuristics.

In the second part of this thesis, we focus on improving the underlying *search algorithms* to increase coverage (the number of tasks solved) and solution quality in planning. We conduct a detailed study of two popular search-control techniques, *preferred operators* and *deferred evaluation*, and demonstrate their respective usefulness for improving coverage and solution quality under various conditions. We also consider *anytime planning* to find high-quality plans given limited time. In anytime planning, the aim is to compute an initial solution quickly, and then iteratively improve on this solution while time remains. We demonstrate that the greediness that is necessary to find an initial plan quickly can impede the planning system in finding better solutions later, unless the system abandons previous effort and restarts the search.

We then combine the methods analysed in the previous chapters and incorporate them into

one planning system. The resulting planner LAMA, winner of the 2008 International Planning Competition, is presented in detail and compared with other state-of-the art planners. We study the interactions of various techniques employed in the system and show how much each feature contributes to the overall performance. We find that both landmarks and restarting anytime search contribute to the good performance on the set of benchmark tasks considered. Furthermore, the two techniques interact beneficially in some cases.

Lastly, we provide an outlook on possible extensions of our work by investigating more complex types of landmarks. We show that using higher-order landmarks can significantly improve the heuristic estimates obtained from a landmark heuristic. However, the additional effort required for finding and using such landmarks does not necessarily pay off.

To my family

Declaration

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Silvia Richter
Brisbane, November 2010

Acknowledgements

This work has benefited greatly from the input and support of many people over the past four years. I would like to express my gratitude to everyone who contributed to it in some way or other.

First and foremost, I would like to thank my academic advisors Charles Gretton, Malte Helmert and Abdul Sattar.

I am deeply grateful to Charles Gretton for his guidance, support and encouragement throughout my PhD candidature. His enthusiasm for research and can-do attitude made working with him an extremely enjoyable experience. Charles was always ready to help, pushing me at the right times, advising me on academic and personal questions alike and always looking out for opportunities for me.

Malte Helmert has had an extraordinary influence on my research. The job I had with him as an undergraduate student first introduced me to planning. Ever since then he has been my academic mentor and his advice has been invaluable to me. I thank Malte for always taking the time to answer my numerous emails, even in extremely busy times, and for his mentoring regarding academic networking, reviewing and similar activities that put me in touch with the scientific community. His willingness to share his knowledge and readiness to help others are unequaled.

I am grateful to Abdul Sattar for making this PhD project possible. Abdul facilitated my scholarship at Griffith University and has given me access to a stimulating working environment at NICTA. He has furthermore provided supervision and mentoring throughout my candidature and beyond, for which I am thankful.

Bernhard Nebel facilitated my first visit to Australia and gave me the opportunity to visit his group in Freiburg several times throughout this PhD project. Thank you for that.

Furthermore, I would like to acknowledge Griffith University and NICTA for their financial support during this study and for providing the budget for several conference travels. Terry Caelli deserves special mention for his mentoring and for making some conference trips possible that could not have taken place otherwise. I also want to thank everyone in the NICTA Queensland lab for making it a great working environment. Special thanks go to Conrad Sanderson, Nathan Robinson, Felix Werner and Ryan Wishart for laughs, interesting discussions and moral support, and to Laurianne Sitbon for many enjoyable walks around the lake and for acting as a replacement supervisor in Charles' absence.

I also want to express a heartfelt thank you to the wonderful AI planning community. I ap-

preciate the fruitful collaborations with Emil Keyder, Jordan Thayer and Wheeler Ruml. A huge thank you goes to Patrik Haslum for providing feedback on a number of my papers, the confirmation report and the thesis itself. Erez Karpas and Peter Hutterer have also proof-read the thesis, for which I am grateful. Furthermore, I very much appreciated the wonderful feedback by my thesis reviewers, Sylvie Thiébaux and Carmel Domshlak. Special thanks also go to Sylvie for her ongoing support and mentoring, and to Gabriele Röger for being a great hotel room mate and exploring Chicago, Thessaloniki and Toronto together. I very much enjoyed my trips to ICAPS and AAAI, and fondly remember the fun nights and interesting discussions with Malte, Gabi, Charles, Erez, Nathan, Sylvie, Carmel and many others.

I thank my friends in Brisbane for the fantastic time we have had, and my friends on the other side of the world for keeping in touch despite the distance. Everyone at Unidive deserves thanks for the great trips that let me see the light when I was cooped up in the office.

Last, but definitely not least, I want to thank my parents Ingeborg and Axel Richter for their constant and unwavering support over all those years.

Contents

List of Figures

List of Tables

List of Publications

During the period of PhD candidature the following publications were authored, and are incorporated into this thesis:

- Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 975–982. AAAI Press, 2008
- Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 273–280. AAAI Press, 2009
- Silvia Richter, Jordan T. Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz, editors, *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 137–144. AAAI Press, 2010
- Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010
- Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for and/or graphs. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 335–340. IOS Press, 2010

1

Introduction

The problem of plan generation in artificial intelligence (AI) consists in computing a sequence of actions that will transform a given initial state of the world into a state that satisfies a specified goal condition. The appeal, but also the challenge of AI planning consists in the aim to develop general problem solvers, i. e., algorithmic approaches that are capable of solving tasks from a wide variety of application areas, requiring only a declarative description of the dynamics of the world. Examples of problems that have been modelled with planning formulations include logistics domains, where the task is to organise the delivery of objects between various locations using a fleet of vehicles; manufacturing domains, where complex structures need to be built from parts according to certain constraints; and action control for autonomous robots such as a Mars rover.

An example of a small planning task from the benchmark domain Logistics (McDermott, 2000) is depicted in Figure 1.1. The "world" consists of a set of locations that are grouped into cities; here, the city on the left contains locations *A* through *D*, while the city on the right contains only one location, *E*. Two types of vehicles exist: trucks and planes. The trucks are constrained to stay within a city and can drive between any two locations in a city. Planes can fly between the airports of different cities. In our example, locations *C* and *E* are airports, denoted by the triangular signs. The goal of a Logistics task is generally to deliver a number of objects to their respective goal locations by transporting them with the available vehicles. In our example, the *box* must be delivered to location *E*, as denoted by the dotted line. In the initial state depicted, the box

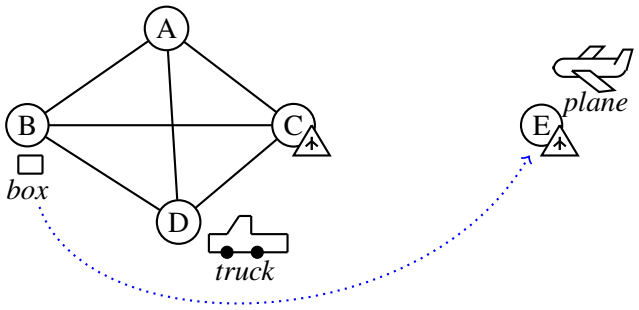


Figure 1.1: A simple Logistics task: transport the box from location *B* to *E*.

is at location *B*, a *truck* is at location *D*, and a *plane* is at *E*. The available actions are to drive the truck between the locations in the left city, to fly the plane between the two airports, and to load and unload the box into or from the truck and/or plane.

A human looking at Figure 1.1 can easily determine that the box will have to be inside the plane at some point, because this is the only way of transporting it from the left city to the right city. Human problem solvers are also typically very good at realising that the order of certain actions is irrelevant (e. g., it does not matter whether we first fly the plane from *E* to *C* or whether we first transport the box to *C* with the truck). To an automated planning system, however, such knowledge is not immediately available, and making this knowledge explicit can notably improve the performance of the system.

A solution (a plan) for a planning task is considered optimal if the number of actions in it is minimal among all possible plans. In the cost-based variant of this classical setting, actions have associated non-negative integer costs, and a plan is considered optimal if the sum of its action costs is minimal among all plans. Both *satisficing planning*, where the task is to find any plan, and *optimal planning*, where plans are required to be optimal, are PSPACE-complete in common planning formalisms (Bylander, 1994). In practice (on typical planning benchmarks) however, satisficing planning is significantly easier than optimal planning (see e. g. Helmert, 2008).

1.1 A Brief History of Planning

Planning has been a core topic of research in artificial intelligence since the beginning of AI in the late 1950s (see e. g. Newell and Simon, 1963; Fikes and Nilsson, 1971). Popular approaches for finding optimal solutions include symbolic exploration of the state space induced by all action sequences (Edelkamp and Helmert, 2001; Edelkamp, 2005), e. g., using binary decision diagrams (Bryant, 1986), and compilation approaches that translate planning tasks into constraint-based formalisms like boolean satisfiability (Kautz and Selman, 1992; Rintanen et al., 2006), constraint networks (van Beek and Chen, 1999) and (mixed) integer programs (Kautz and Walser, 1999; Benton et al., 2007). The most widely used approach to planning today, however, is heuristic search in the state space. In particular in satisficing planning, where solutions are not required to be optimal but the aim is to find a plan of reasonable quality quickly, state-space heuristic search has outperformed other approaches by a large margin in the past decade. Since 2000, for example, all winners of the classical satisficing track in the roughly biennial International Planning Competition (short IPC, see e. g. McDermott, 2000; Helmert et al., 2008) have followed this paradigm (Hoffmann and Nebel, 2001; Gerevini et al., 2003; Helmert, 2006; Chen et al., 2006; Richter and Westphal, 2008). Recent developments, however, suggest that this picture may change. (See e. g. Rintanen, 2010, for a highly competitive planner based on satisfiability.) In the past two years, state-space heuristic search has also become the state of the art for optimal planning (Karpas and Domshlak, 2009; Helmert and Domshlak, 2009).

The commonly used *forward search* explores the space of all reachable world states by itera-

tively choosing a reached state and *expanding* it, i. e., computing its successor states and adding them to the search space, until either a state is found that satisfies the goal conditions or all reachable states have been exhausted. (In the latter case the planning task has no solution.) The choice of which state to expand next is based on a heuristic function that estimates how far a given state is from a goal state. Planning systems mainly differ in their use of heuristic functions and their choice of search algorithm. For example, greedy best-first search (Pearl, 1984) typically leads to finding a (possibly suboptimal) solution quickly, whereas the A* algorithm (Hart et al., 1968) guarantees optimal solutions when used with heuristics that have certain properties. However, A* usually needs more time. In the following, we briefly discuss a number of concepts that are central to the contributions in this thesis.

Heuristics. The central problem in developing heuristic search approaches to planning is computing a suitable heuristic function for estimating goal distance. Major milestones in the history of satisficing heuristic-search planning are the *additive heuristic* (Bonet and Geffner, 2001), the very influential *FF heuristic* (Hoffmann and Nebel, 2001), and the *context-enhanced additive heuristic* (Helmert and Geffner, 2008). For optimal planning, important contributions have been the *max heuristic* (Bonet and Geffner, 2001), the family of h^m *heuristics* (Haslum and Geffner, 2000), the family of *pattern database heuristics* (Edelkamp, 2001), *merge-and-shrink heuristics* (Helmert et al., 2007), and, most recently, *landmark heuristics* based partially on work presented in this thesis (Karpas and Domshlak, 2009; Helmert and Domshlak, 2009).

Most of these heuristics are computed by solving a *simplified* planning task for a given state and using the solution cost of this simplified task as an indication of the solution cost of the actual task. In the case of the FF heuristic, for example, negative effects of actions such as the consumption of resources are ignored.

Search enhancements. In addition to the heuristic cost estimate, most successful planning systems exploit further information about the solution of the simplified task. The FF planning system (Hoffmann and Nebel, 2001), winner of the classical track at the International Planning Competition (IPC) in 2000, computes so-called *helpful actions* that contribute to the solution of the simplified task. These actions are treated preferentially in the underlying search algorithm, since often actions that contribute to solving the simplified task are also useful for solving the actual task. The Fast Downward planning system (Helmert, 2006), winner of the satisficing track at IPC 2004, generalises the idea of helpful actions to other heuristics and integrates them under the name *preferred operators* into greedy best-first search. In addition, Fast Downward uses a search enhancement technique called *deferred evaluation* that postpones the computationally costly heuristic evaluation of states.

Control knowledge. Since the early days of planning research, planning systems have also been supplied with so-called *control knowledge* to aid the search for a solution. Control knowledge is

knowledge about the structure of solutions or promising solution strategies. A planning system can learn this knowledge from previously solved tasks or derive it from the specification of the task at hand. Examples of control knowledge include knowledge about *symmetries* in the planning task (Fox and Long, 1999, 2002), and about sequences of actions that often must be executed in order, so-called *macros* (Fikes et al., 1972; Botea et al., 2005). Such control knowledge can be used to enrich the task description, making the knowledge explicitly accessible to the search algorithm. A type of control knowledge that is of particular importance to this work is *landmarks* (Porteous et al., 2001; Hoffmann et al., 2004). Landmarks describe properties of states that must be traversed in every plan, i. e., for every landmark it holds that every solution for the planning task at hand must traverse a state with the given property. Landmarks have been previously used to simplify planning by partitioning the task into smaller subtasks, ordering the landmarks and then achieving them one at a time (Hoffmann et al., 2004; Sebastia et al., 2006).

Solution quality. To date, not many satisficing planning systems aim to achieve high-quality solutions. Popular systems like FF and Fast Downward, winners of past IPCs, try to find a solution as quickly as possible with no concern for the quality of that solution. In this work, we are particularly interested in finding plans of high quality quickly. One special case of this is in the setting of cost-based planning, where we reason about costs that are associated with the actions in a plan. We believe that emphasising high plan quality is an important trait for a planner when aiming to solve "real-world" problems.

Anytime algorithms. Methods that aim to deliver the best-possible solution at any given point in time are called anytime algorithms. Rather than terminating upon finding a solution, anytime algorithms compute a first (typically suboptimal) solution as quickly as possible, and then search for iteratively better solutions until they are interrupted. Various anytime algorithms for heuristic search have been proposed (Likhachev et al., 2004; Hansen and Zhou, 2007; Aine et al., 2007), but they have typically not been applied to planning.

1.2 Outline and Contributions of this Thesis

In this work, we propose various heuristics and algorithms aimed at quickly finding high-quality plans. Building on these methods, we design and implement a state-of-the-art planning system, the LAMA planner, that successfully trades efficiency (i. e., plan generation time) against solution quality. We conduct extensive studies of some of the design choices in this planner, leading to findings that will be of benefit to research within and beyond planning. Below, we provide an outline of the structure of this thesis and its contributions.

After giving a background on existing heuristic-search techniques for planning in Chapter 2, the first part of this thesis (Chapters 3–5) proposes several advances related to using landmarks in planning. We introduce novel methods for finding landmarks in a given planning task, and

propose using landmarks as a *heuristic function*. We evaluate the benefit of using landmarks in this way in satisficing planning, both with and without action costs. In the second part of this thesis, (Chapters 6 and 7), we examine improvements to search algorithms for planning. We contrast different ways of using the search enhancements *preferred operators* and *deferred evaluation*, and study their merits and interactions. In order to deliver high-quality solutions given an unknown time horizon, we propose a novel anytime algorithm for planning that avoids over-commitment due to early greediness. In the third part of this thesis (Chapter 8), we put the pieces together and present a system that integrates the set of techniques discussed previously, the LAMA planner. This system, which won the International Planning Competition in 2008, is contrasted against other planners and evaluated experimentally, analysing how its various features work together to achieve good performance on the set of planning benchmarks from the competition. Finally, we give an outlook on possible extensions of our work by investigating the benefit of *higher-order landmarks* in Chapter 9. We conclude in Chapter 10, review work by other authors that builds on the material presented here, and identify possible directions of further research. Below, we outline the chapters containing novel contributions in some more detail.

Finding Fact Landmarks and Disjunctive Landmarks (Chapter 3)

We start by introducing a novel method for deriving landmarks from *domain transition graphs*. This approach proceeds by translating planning tasks into a finite-domain state variable representation and exploiting the information made explicit by that representation. Together with previous approaches that we extend to (a) deal with our representation and (b) also produce *disjunctive* landmarks, we arrive at a method that gives us a rich set of landmarks that will prove useful in the subsequent chapters of this thesis. We describe the resulting method in detail and contrast the number and types of landmarks it finds with those of previous approaches.

Using Landmarks as a Heuristic (Chapter 4)

Next, we propose a new way of *using* landmarks by deriving a heuristic function from them, counting for each state how many landmarks must be achieved between that state and a goal. The integration of this information with a base heuristic (the FF heuristic) is shown to improve the number of tasks solved and the solution qualities of a planner, compared to only using the base heuristic. While by itself the landmark heuristic is not competitive with the base heuristic, the combination of the two heuristics results in notable improvement. This shows that control knowledge in the form of landmarks can be usefully employed as additional information within a heuristic search algorithm.

Landmarks in Cost-Sensitive Planning (Chapter 5)

When using the combination of the FF heuristic and landmark heuristic discussed in Chapter 4 in a *cost-based* setting, the performance gain achieved by using landmarks is substantially higher than

in the traditional setting of planning without costs. This is mainly due to bad performance of the FF heuristic in cost-based planning, a problem that landmarks help to mitigate. In this chapter, we analyse why the FF heuristic may perform badly in settings with action costs and how landmarks help to overcome the problem.

Preferred Operators and Deferred Evaluation (Chapter 6)

In the second part of this thesis, we look at improvements to search algorithms. Preferred operators and deferred evaluation are two techniques that have been employed by several planning systems with the aim of speeding up the search. Despite the widespread use of these search-enhancement techniques however, few results have been previously published detailing their usefulness. In particular, while various ways of using, and possibly combining, these techniques are conceivable, no previous work has studied the performance of such variations. In this chapter, we address this gap by examining the use of preferred operators and deferred evaluation in a variety of settings within best-first search. In particular, our findings are consistent with and help explain the good performance of the winner of the propositional satisficing track at IPC 2004.

Restarts for Anytime Planning (Chapter 7)

In order to deliver an initial solution quickly, anytime search algorithms are typically greedy with respect to the heuristic cost-to-go estimate h. In this chapter, we show that this low-h bias can cause poor performance if the greedy search makes early mistakes. Building on this observation, we present a new anytime approach that restarts the search from the initial state every time a new solution is found. We demonstrate the utility of our method via experiments in planning as well as other domains and show that it is particularly useful for problems where the heuristic has systematic errors.

The LAMA Planner (Chapter 8)

LAMA is a classical planning system that incorporates the techniques discussed in the previous chapters. It finds a first solution quickly using greedy best-first search and then improves on this solution using restarting anytime search as long as time remains. It uses landmarks in addition to the FF heuristic, and employs the search enhancements preferred operators and deferred evaluation in their most effective combination.

LAMA showed best performance among all planners in the sequential satisficing track of the International Planning Competition in 2008. In this chapter, we present the system in detail and investigate which features of LAMA are crucial to its performance, and how they interact. We find that the landmark heuristic and the restarting anytime search both contribute to improving the results, and there are even synergy effects between them. While the incorporation of action costs into the heuristic estimators, as discussed in Chapter 5, proved not to be beneficial on the competition benchmarks, the landmarks mitigate the bad performance of the cost-sensitive FF

heuristic in LAMA and led it to outperform the other cost-sensitive planners at IPC 2008. We also show results on traditional benchmarks without action costs, demonstrating that LAMA is competitive with the state of the art in satisficing planning, in particular if we are interested in good solution quality.

Sound and Complete Landmarks for And/Or Graphs (Chapter 9)

In this chapter, we take one step into the direction of future work by analysing an approach for identifying landmarks that allows us to find *conjunctive landmarks*. Along the way, we show how this approach can be used to find landmarks for general *and/or graphs*, a generalisation compared to the *relaxed planning graphs* used to find landmarks for planning tasks. We demonstrate that this method finds strictly more *causal* landmarks than previous approaches but is costly to compute. Building on this, we discuss the relationship between increased computational effort for landmark discovery and experimental performance of a planner using these landmarks in a landmark-counting heuristic.

2

Background

In this chapter we review formalisms, algorithms, heuristics and search enhancements for automated planning via state-space search. The concepts described here form an essential background for large parts of this thesis, whereas related work that concerns only particular chapters is discussed directly in those chapters.

2.1 The Planning Problem

Planning tasks are formalised using *state variables* that can take on values from a finite domain, and *actions* that describe how the values of the state variables change when the action is applied. The task consists of finding a sequence of actions that, when applied in order, transform a specified initial state into a state satisfying given goal conditions. In the traditional *STRIPS* formalism for planning, named after an early planning system (Fikes and Nilsson, 1971), the state variables are of binary range. In recent years, models allowing arbitrary finite domains have become popular. One such formalism, the SAS$^+$ planning model (Bäckström and Nebel, 1995), will be used in this thesis. The following definition has been adapted from Helmert (2006).

Definition 2.1. ***SAS$^+$ planning tasks***
*An **SAS$^+$ planning task** is given by a 4-tuple $\langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$ with the following components:*

- *$\mathcal{V}$ is a finite set of **state variables**, each with an associated finite domain $\mathcal{D}_v$. A **fact** is a pair $\langle v, d \rangle$ (also written $v \mapsto d$), where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. A **partial variable assignment** s is a set of facts, each with a different variable. (We use set notation such as $\langle v, d \rangle \in s$ and function notation such as $s(v) = d$ interchangeably.) A **state** is a variable assignment defined on all variables in $\mathcal{V}$.*
- *s_0 is a state called the **initial state**.*
- *$s_\star$ is a partial variable assignment called the **goal**.*
- *$\mathcal{A}$ is a finite set of **actions**, each with two associated partial variable assignments pre(a) and eff(a), where the facts in pre(a) and eff(a) are called the **preconditions** and* effects *of*

the action, respectively. Each action furthermore has an associated non-negative integer $cost(a)$ *called the* cost *of the action.*

An action $a \in \mathcal{A}$ *is* ***applicable*** *in a state* s *if* $pre(a) \subseteq s$*. In this case, we say that the action can be* ***applied*** *to* s *resulting in the state* s' *with* $s'(v) = \mathit{eff}(a)(v)$ *where* $\mathit{eff}(a)(v)$ *is defined and* $s'(v) = s(v)$ *otherwise. We write* $s[a]$ *for* s'*. For action sequences* $\pi = \langle a_1, \ldots, a_n \rangle$*, we write* $s[\pi]$ *for* $s[a_1] \ldots [a_n]$ *(only defined if each action is applicable in the respective state).*

An action sequence π *is called a* ***plan*** *if* $s_\star \subseteq s_0[\pi]$*. The* ***cost*** *of* π *is the sum of the costs of its actions,* $\sum_{i=1}^{n} cost(a_i)$*. The* ***satisficing planning problem*** *consists in finding a plan for a given SAS*$^+$ *task or showing that no such plan exists. The* ***optimal planning problem*** *consists in finding a plan of minimal cost for a given SAS*$^+$ *task or showing that no such plan exists.*

In traditional (unit-cost) planning, all actions have an associated cost of 1. We will in the following assume unit-cost planning unless otherwise noted, and refer to the alternative as *cost-based* planning. Figure 2.1 shows part of an SAS$^+$ encoding of the example task that was given in Figure 1.1 in the introduction.

$$
\begin{aligned}
\mathcal{V} &= \{v_\mathrm{t}, v_\mathrm{p}, v_\mathrm{b}\} \\
\mathcal{D}_{v_\mathrm{t}} &= \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\} \\
\mathcal{D}_{v_\mathrm{p}} &= \{\mathrm{C}, \mathrm{E}\} \\
\mathcal{D}_{v_\mathrm{b}} &= \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{t}, \mathrm{p}\} \\
\mathcal{A} &= \{a_1, a_2, a_3, \ldots \mid \\
&\qquad \mathit{pre}(a_1) = \{v_\mathrm{t} \mapsto \mathrm{A}\}, \quad \mathit{eff}(a_1) = \{v_\mathrm{t} \mapsto \mathrm{B}\}, \quad \mathit{cost}(a_1) = 1 \\
&\qquad \mathit{pre}(a_2) = \{v_\mathrm{t} \mapsto \mathrm{B}, v_\mathrm{b} \mapsto \mathrm{B}\}, \quad \mathit{eff}(a_2) = \{v_\mathrm{b} \mapsto \mathrm{t}\}, \quad \mathit{cost}(a_2) = 1 \\
&\qquad \mathit{pre}(a_3) = \{v_\mathrm{p} \mapsto \mathrm{E}, v_\mathrm{b} \mapsto \mathrm{p}\}, \quad \mathit{eff}(a_3) = \{v_\mathrm{b} \mapsto \mathrm{E}\}, \quad \mathit{cost}(a_3) = 1\} \\
s_0 &= \{v_t \mapsto \mathrm{D}, v_\mathrm{p} \mapsto \mathrm{E}, v_\mathrm{b} \mapsto \mathrm{B}\} \\
s_\star &= \{v_\mathrm{b} \mapsto \mathrm{E}\}
\end{aligned}
$$

Figure 2.1: Partial SAS$^+$ encoding of the example task from Figure 1.1. Three actions are shown, one instance each of a drive, load and unload action.

STRIPS and the delete relaxation. The STRIPS formalism mentioned earlier is a special case of the SAS$^+$ formalism where variables are binary (taking on the values True and False), and the value True may appear in action preconditions and the goal. The effects of actions can be partitioned into *add effects* that assign True, and *delete effects* that assign False to the corresponding variables. A popular technique to conduct a simplified reachability analysis of the facts in a task is to ignore the delete effects of actions, meaning that facts that are true before applying an action will not be made false by applying the action (Bonet and Geffner, 2001). For example, after driving the truck in our Logistics example from *A* to *B*, the truck will be at both locations simultaneously

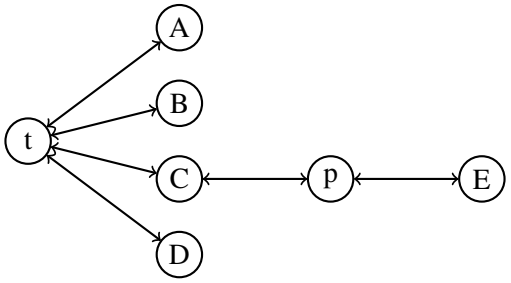


Figure 2.2: Domain transition graph for the location of the box in our example task (Figure 1.1).

in the relaxed task. Any action in the relaxed task can thus only *add* facts (make them true), not remove them (make them false), and the set of true facts increases monotonically from the initial state onwards. A sequence of actions is a plan for the relaxed task if it adds all goal facts. In the STRIPS formalism, this so-called *delete relaxation* of a task is obtained from the original task by setting the delete effects of all actions to the empty set. For the SAS$^+$ model, we formalise the delete relaxation as follows.

Definition 2.2. ***The Delete Relaxation***
The ***delete relaxation*** *of an SAS$^+$ planning task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$ is the SAS$^+$ task Π^+ with $\Pi^+ = \langle \mathcal{V}^+, s_0^+, s_\star^+, \mathcal{A}^+ \rangle$, where $\mathcal{V}^+ = \{v_d \mid v \in \mathcal{V}, d \in \mathcal{D}_v\}$ with $\mathcal{D}_v = \{reached, unreached\}$ for $v \in \mathcal{V}^+$, $s_0^+ = \{v_d \mapsto reached \mid v \mapsto d \in s_0\} \cup \{v_d \mapsto unreached \mid v \mapsto d' \in s_0, d \neq d'\}$, $s_\star^+ = \{v_d \mapsto reached \mid v \mapsto d \in s_\star\}$, and $A^+ = \{a^+ \mid a \in A\}$, where $pre(a^+) = \{v_d \mapsto reached \mid v \mapsto d \in pre(a)\}$, $\mathit{eff}(a^+) = \{v_d \mapsto reached \mid v \mapsto d \in \mathit{eff}(a)\}$, and $cost(a^+) = cost(a)$.*

A useful tool for making the structure of SAS$^+$ planning tasks explicit is the *domain transition graph* that captures the ways in which the value of a given state variable v may change (Jonsson and Bäckström, 1998; Helmert, 2006). It is a directed graph with vertex set $\mathcal{D}_v$ that contains an arc between two nodes d and d' if there exists an action that can change the value of v from d to d'. Formally (adapted from Helmert, 2004):

Definition 2.3. ***Domain transition graphs***
The ***domain transition graph (DTG)*** *of a state variable $v \in \mathcal{V}$ of an SAS$^+$ task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$ is the digraph $\langle \mathcal{D}_v, A \rangle$ which includes an arc $\langle d, d' \rangle$ iff the following conditions hold: $d \neq d'$ and there is an action $a \in \mathcal{A}$ with $\mathit{eff}(a)(v) = d'$ and either $pre(a)(v) = d$ or $pre(a)(v)$ is undefined.*

Figure 2.2 shows the domain transition graph for the box in our example task from Figure 1.1. It can be seen from the domain transition graph that, in order to transport the box from A to E, it will have to be in the truck, at C, and in the plane, in sequence. One use of domain transition graphs has been to compute heuristic goal distance estimates (see Section 2.2.2).

PDDL. The encoding language for planning tasks is usually PDDL (McDermott, 2000; Fox and Long, 2003; Edelkamp and Hoffmann, 2004), which has been used for the International Planning

Competitions held since 1998 (see Section 2.4). PDDL allows representing planning tasks concisely using *first-order literals* and logical connectives. It also has many advanced features such as actions with *conditional effects*, and *axioms* (or "derived predicates") that prescribe automatic value changes of special state variables that are not directly set by actions, but depend on the values of other variables. The base of PDDL is a restricted first-order language $\mathcal{L}$ with finitely many predicate symbols, constants and variables. States are sets of ground atoms from $\mathcal{L}$, while actions are represented by *planning operators*. Operators contain variable symbols and can be *instantiated* or *grounded* by substituting the variable symbols with constants from $\mathcal{L}$. An action is any ground instance of a planning operator. For example, an operator "drive from l_1 to l_2", where l_1 and l_2 are variables, may be instantiated in a planning system to yield drive actions for all pairs of locations within a city. Planning in this first-order formalism is EXPSPACE-complete (Erol et al., 1995). Most planners do not use the PDDL representation internally, but translate PDDL tasks into simpler formalisms and ground all operators before planning.

The Fast Downward planning system. The implementations and experimental evaluations undertaken for this thesis are all based on the Fast Downward planner (Helmert and Richter, 2004; Helmert, 2006). Fast Downward compiles PDDL input into a ground representation with variables of arbitrary finite range (Helmert, 2009). The representation used in Fast Downward is based on SAS$^+$, but extends it with conditional effects and derived predicates. These features of Fast Downward are orthogonal to our contributions. In order to keep our presentation simple, we will use the SAS$^+$ formalism here. However, in those cases where the implementations support advanced features in our experimental evaluations, we will include benchmark tasks having these features in our results.

2.2 Planning by Heuristic Search

Heuristic search is a widely used framework for solving shortest-path problems. In planning, a heuristic forward search proceeds by iteratively *expanding* a given *search node* corresponding to a state (starting with the initial state) by applying all possible actions in the state. The resulting successor states are evaluated using a heuristic function h. This function estimates the cost of a minimum-cost path between the given state and a goal state (which in unit-cost planning is equivalent to the *distance* to a nearest goal state). At each subsequent iteration, the search chooses the next node (state) to be expanded based on the heuristic values computed previously. A heuristic may be any function that maps states to non-negative numbers and assigns the heuristic value of zero to goal states. In the following, we review common search algorithms and heuristics used in planning, as well as so-called *search-enhancement techniques*, i. e. modifications and additions to the standard search algorithms that have been used to improve planning performance.

2.2.1 Algorithms

The methods we discuss here all conduct forward search in the state space of the planning task. We discuss two methods that have been successful in satisficing planning, *greedy best-first search* and the *enforced hill-climbing* algorithm. We review the A^* algorithm as the dominant approach for optimal planning and briefly discuss *weighted* A^* as a variant that allows trading solution quality for speed.

With the exception of enforced hill-climbing, all of these algorithms are global *best-first search* algorithms and maintain two data structures for storing search nodes, the open list and the closed list. The open list contains nodes that have been encountered but not yet expanded, whereas the closed list contains nodes that have already been expanded. In the beginning, the open list contains only a node corresponding to the initial state and the closed list is empty. Iteratively one of the nodes from the open list is selected, moved from the open list to the closed list, and expanded, i. e., the successor states of the corresponding state are computed. These successors are then evaluated using a heuristic function and inserted as nodes into the open list along with their heuristic values. The algorithms differ in how they select the next node to be expanded and in what they do when encountering nodes that correspond to states that have already been expanded. We equate search nodes with their corresponding states in the descriptions below.

Greedy Best-First Search

The simplest of the global heuristic-search algorithms is greedy best-first search (Pearl, 1984), which always expands a state with *lowest heuristic value* among all states in the open list, i. e., a state that is estimated to be closest (in terms of path length or cost) to a goal. Standard implementations of this algorithm never process a state more than once, so that if a state is encountered that is already present in the closed list, the search simply ignores this state, even if the state has been reached on a shorter/cheaper path the second time. Overall, greedy best-first search is not concerned with plan length or cost, but aims to find a solution as quickly as possible. Given a finite number of states as in planning, this algorithm is complete since it will eventually exhaust the entire search space. Greedy best-first search is the search algorithm used in the Fast Downward planner.

A* and Weighted A* Search

The A* algorithm (Hart et al., 1968) expands at each step a state that minimises the function $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the best path currently known from the initial state to state s, and $h(s)$ is the heuristic value of s. Among states of equal f-value, the search prefers states with lower heuristic value.

A* is often used with *consistent* heuristics, i. e., ones that satisfy the triangle inequality $h(s) \leq c(s, s') + h(s')$ for all s and s', where $c(s, s')$ is the cost of reaching s' from s. With consistent heuristics, it is guaranteed that when a state is expanded, it has been reached via a minimum-cost

path. Consequently, A* never needs to expand a state more than once and when a goal state is selected for expansion, an optimal solution is found. When using *inconsistent* heuristics in A*, however, it is possible that the search later encounters a cheaper path to a state that has already been expanded. As long as the heuristic function is *admissible*, i. e., never overestimates the true goal distance, it is still possible to preserve optimality of the algorithm by re-expanding such states and propagating the cheaper costs to the successor states.

A* can be modified to trade solution quality for speed by weighting the heuristic with a factor $w > 1$ (Pohl, 1970), thus using the selection function $f'(s) = g(s) + w \cdot h(s)$. The resulting algorithm, called *weighted A**, searches more greedily the larger w is. This typically speeds up the search, but means that solutions are not guaranteed to be optimal any more, since the weighting by w introduces inadmissibility even if the heuristic function h is admissible. However, the solution is guaranteed to be *w-admissible*, i. e. the ratio between its cost and the cost of an optimal solution is bounded by w. In Chapter 7 we present an *anytime algorithm* for planning based on weighted A*.

Enforced Hill-Climbing

Enforced hill-climbing (Hoffmann and Nebel, 2001) is a *local* search algorithm that does not maintain an open list in the way the algorithms described above do. Instead, enforced hill-climbing iteratively tries to find a state of lower heuristic value than its *current state* (starting with the initial state), and once it has found such a state, commits to it by making it the new current state. In each iteration, the approach conducts a *breadth-first search* from the current state until an improved state is found. While this algorithm is often very effective for planning benchmarks and is used in the FF planner (Hoffmann and Nebel, 2001), one of its disadvantages is that due to its local nature it may become trapped in dead ends of the search space. In contrast to the greedy best-first search and the A* search, enforced hill-climbing is thus not complete. The FF planner restarts using a greedy breadth-first search if enforced hill-climbing fails.

2.2.2 Heuristics

A common way to derive heuristics is to simplify a planning task by abstracting away some details, to solve the simplified task, and to use the cost of that solution as a heuristic for the original task. Bonet and Geffner (2001) introduced the delete relaxation given in Definition 2.2 as a way of simplifying planning tasks by ignoring the delete effects of actions. The cost of an optimal plan for the relaxed task is an admissible heuristic for the original task and is called the h^+ heuristic (Hoffmann, 2005). Computing this value exactly, however, is still NP-hard (Bylander, 1994). Therefore, various techniques have been proposed for approximating the h^+ heuristic. A useful structure in this context is the *relaxed planning graph* (Hoffmann and Nebel, 2001), which combines the idea of the delete relaxation with the *planning graph* proposed by Blum and Furst (1997).

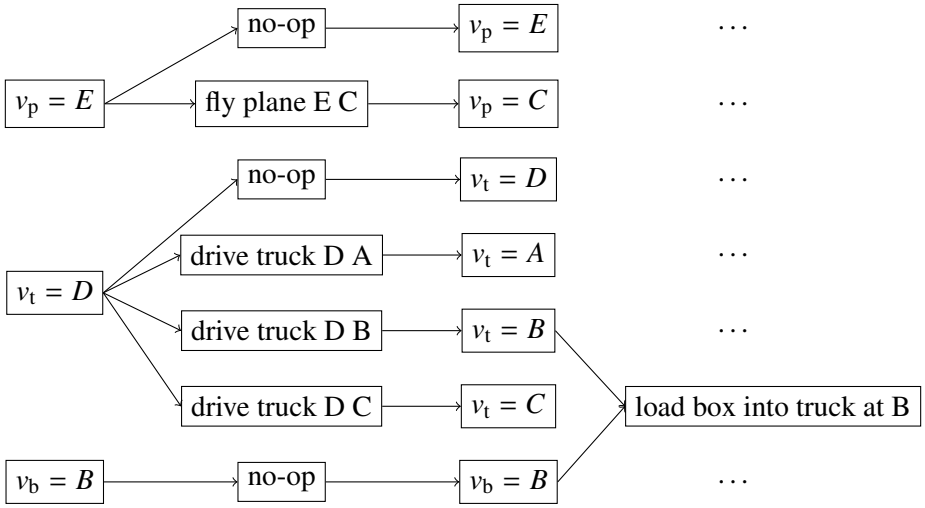


Figure 2.3: Partial relaxed planning graph for our example task (Figure 1.1). Actions are referred to using natural-language descriptions.

The Relaxed Planning Graph

A relaxed planning graph is a directed layered graph that contains two kinds of nodes, *fact nodes* and *action nodes*. The layers of the graph alternate between fact and action layers, each layer containing nodes of the respective kind. A fact layer and the following action layer together make up a *time step*. The first fact layer consists of all the facts that are true in the initial state, and the first action layer consists of all actions that are applicable in the initial state. In each subsequent time step i, the fact layer contains all facts that can possibly be made true in i time steps, and the action layer contains all actions that are possibly applicable given the facts at time step i (Hoffmann and Nebel, 2001). For example, a fact may exist in time step i if it is made true (added) by an action at time step $i - 1$ (even if it is also made false (deleted) by some other action at time step $i - 1$). Thus, the ith fact layer S_i is an over-approximation of all facts that can be reached by applying up to i actions. Similarly, the ith action layer O_i is an over-approximation of all actions that can be applied as the ith action in a plan. Each action node at layer i is connected via edges to its preconditions at layer i and to its effects at layer $i + 1$. The appeal of this structure is that it can be computed very efficiently (in time that is polynomial in the number of facts and actions), yet it captures important information about the structure of possible plans for the relaxed task. In fact, a (non-optimal) solution for the relaxed task can easily be extracted from the graph (Hoffmann and Nebel, 2001), leading to the FF heuristic discussed in Section 2.2.2. Figure 2.3 shows a partial relaxed planning graph for our example task from Figure 1.1, using the encoding given in Figure 2.1. Special "no-op" actions are traditionally used to denote the propagation of facts that have been added at some time step to later time steps. A formal definition of the layers S_i and O_i (not including no-op actions) that make up the relaxed planning graph for a planning

task $\langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$ is as follows, where $i \in \mathbb{N}_0$:

$$S_i := \begin{cases} s_0 & i = 0 \\ S_{i-1} \cup \bigcup_{a \in O_{i-1}} \mathit{eff}(a) & i > 0 \end{cases}$$
$$O_i := \{a \in \mathcal{A} \mid \mathit{pre}(a) \subseteq S_i\}$$

The sets S_i and O_i grow monotonically, reaching a fixed point after finitely many steps due to the number of facts in a planning task being finite. We say that the relaxed planning graph has levelled off at this point and call this last layer of the graph its *fixpoint layer*.

The Max Heuristic and the Additive Heuristic

The max heuristic $h^{\max}$ and the additive heuristic h^{add} (Bonet and Geffner, 2001) are two heuristics that approximate the perfect delete-relaxation heuristic h^+ and that can be derived from the relaxed planning graph (though they were not originally proposed using that terminology). The additive heuristic recursively estimates the cost of achieving a set of facts in a planning task as the *sum* of the estimated costs of the facts in the set. The cost of achieving a fact is determined by the recursive cost of its *best support*, i. e., an action that makes the fact true and has minimal estimated recursive cost among all such actions. The estimated recursive cost of an action is determined in turn by the estimated costs of its preconditions. We give a formal definition of the h^{add} heuristic for a state s below. In order for this definition to be well-defined, we restrict the task to facts that are *relaxed reachable* from s, i. e., facts that appear in the fixpoint layer of the relaxed planning graph using s as the initial state. The heuristic value of s is defined as infinity if any goal fact is not relaxed reachable from s, and we remove all actions that have preconditions that are not relaxed reachable from s. (This transformation does not affect the space of solutions starting from s.) Then, the h^{add} heuristic is defined as follows, where A_f is the set of actions that can achieve a fact f, and F is a set of facts:

$$\begin{aligned}
h^{\text{add}}(s) &\stackrel{\text{def}}{=} h^{\text{add}}(s_\star|s) \\
h^{\text{add}}(F|s) &\stackrel{\text{def}}{=} \sum_{f \in F} h^{\text{add}}(f|s) \\
h^{\text{add}}(f|s) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f \in s \\ h^{\text{add}}(\mathit{supp}(f|s)|s) & \text{otherwise} \end{cases} \\
\mathit{supp}(f|s) &\stackrel{\text{def}}{=} \text{argmin}_{a \in A_f} h^{\text{add}}(a|s) \\
h^{\text{add}}(a|s) &\stackrel{\text{def}}{=} \mathit{cost}(a) + h^{\text{add}}(\mathit{pre}(a)|s)
\end{aligned} \tag{2.1}$$

The definition of the max heuristic is analogous, with the cost of achieving a set of facts being estimated by the *maximum*, rather than the sum, of the costs of the individual facts. For both the additive and the max heuristic, the costs for facts and actions that make up a heuristic value may

be calculated in a forward manner by treating the recursive equations as update rules and applying them iteratively until reaching a fixed point. This can be viewed conceptually as propagating cost information for actions and facts along the edges of a relaxed planning graph – propagating costs from preconditions to applicable actions and from actions to effects – in accordance with the equations. However, the relaxed planning graph does not need to be explicitly constructed. Instead, a form of generalised Dijkstra shortest-path algorithm as described by Liu et al. (2002) may be used, which represents each action and fact only once. This reduces the time and space requirements from roughly quadratic ($O(N^2 \log N)$, where N is the encoding size of the task) to roughly linear ($O(N \log N)$).

While the max heuristic is admissible and may thus be used in an A* algorithm for optimal planning, the inadmissible additive heuristic is the one showing better performance in the context of satisficing planning (Bonet and Geffner, 2001).

The FF Heuristic and the FF/add Heuristic

The FF heuristic (Hoffmann and Nebel, 2001) was proposed for unit-cost planning and extracts a (non-optimal) plan for the relaxed task from the relaxed planning graph, using the length of that relaxed plan to estimate the distance to the goal in the original task. The heuristic is computed in two phases for a given state s: the first phase, or *forward phase*, builds the graph and uses it to calculate an estimate, for each fact and action in the planning task, of how costly it is to achieve from s in a relaxed task. The cost of achieving a fact or action is estimated by the index of the *time step* in which it first occurs in the graph. Concurrently, a best support is selected for each fact F, i. e., a cheapest achieving action with regard to the cost estimates. (Ties are broken using additional criteria.) In the second phase, a *plan* for the relaxed task is computed based on the best supports for each fact. This is done by chaining backwards from the goals, selecting the best supports of the goals, and then recursively selecting the best supports for the preconditions of already selected actions. The union of these best supports constitutes the relaxed plan (i. e., for each fact its best support is added only once to the relaxed plan, even if the fact is needed several times as a precondition). The length of the resulting relaxed plan is the heuristic estimate reported for s.

As with the max heuristic and the additive heuristic, the forward phase can be viewed as propagating cost information for actions and facts in a relaxed planning graph, or equivalently, performing cost propagation using a generalised Dijkstra cheapest-path algorithm. As described above, the cost estimate for an action in the FF heuristic is its *depth* in the relaxed planning graph, which in the case of planning with unit-cost actions is equivalent (Fuentetaja et al., 2009) to propagating costs via the $h^{\max}$ criterion. (See Fuentetaja et al., 2009, for a detailed description of how the FF heuristic and the max heuristic relate.) Using other criteria for cost propagation results in variations of the FF heuristic (Bryce and Kambhampati, 2007; Fuentetaja et al., 2009). One variant that has been previously proposed (Do and Kambhampati, 2003; Keyder and Geffner, 2008), in particular for cost-based planning, is to use the h^{add} criterion. We will use the term

FF/add for this variant of the FF heuristic. A formal specification of the cost-sensitive FF/add heuristic is then given by

$$
\begin{aligned}
h^{\mathrm{FF/add}}(s) &\stackrel{\text{def}}{=} \sum_{a \in \pi(s)} cost(a) \qquad (2.2)\\
\pi(s) &\stackrel{\text{def}}{=} \pi(s_\star|s)\\
\pi(F|s) &\stackrel{\text{def}}{=} \bigcup_{f \in F} \pi(f|s)\\
\pi(f|s) &\stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } f \in s \\ \{supp(f|s)\} \cup \bigcup_{\tilde{f} \in pre(supp(f|s))} \pi(\tilde{f}|s) & \text{otherwise} \end{cases}
\end{aligned}
$$

where $supp(f|s)$ is the best support of f according to Equation 2.1. If at any point in the recursion $supp(f|s)$ is undefined because no achieving action exists for f, $h^{\mathrm{FF/add}}(s)$ is infinite. A relaxed plan for the task can be formed by ordering the actions in $\pi(s)$ appropriately. A cost-unaware version of this heuristic, estimating the *distance* rather than the *cost* to a closest goal, is obtained by using the length of the relaxed plan $|\pi(s)|$ (as in the original FF heuristic) rather than the sum of the action costs $\sum_{a \in \pi(s)} cost(a)$ in Equation 2.2.

The FF/add heuristic is used in several of our experimental evaluations. Details about its performance in cost-based planning are discussed in Chapter 5.

The Causal Graph Heuristic and Context-Enhanced Additive Heuristic

The causal-graph heuristic (Helmert, 2004) was specifically designed for planning with state variables that take on values from an arbitrary finite rather than binary domain. In order to convey its underlying idea, we first introduce the *causal graph* which describes the dependencies between the state variables in a planning task.

Definition 2.4. ***Causal Graph***
*We say that an action a **affects** a variable v if $v \mapsto d \in eff(a)$ for some $d \in \mathcal{D}_v$. We say that an action a is **preconditioned** on a variable v if $v \mapsto d \in pre(a)$ for some $d \in \mathcal{D}_v$.*

*The **causal graph** of a planning task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$ is a directed graph with vertex set $\mathcal{V}$, that contains the arc (v, v') for $v \neq v'$ iff there is an action that affects v' and is preconditioned on v, or there is an action that affects both v and v'.*

Like the additive heuristic, the causal-graph heuristic estimates the cost of reaching the goal condition $s_\star$ from a given state s as the sum of the costs, over all variables v that form part of $s_\star$, of achieving the goal value $s_\star(v)$ from the current value $s(v)$. While the additive heuristic assumes all state variables to be independent of each other, the causal-graph heuristic takes into account some of the dependencies between the variables as given by the causal graph. If a state variable v is a *root node* in the causal graph of the task, any action affecting v is not preconditioned on another variable $v' \neq v$, i. e., the value of v can be changed independently of other variables. This means the cost $c(v, d, d')$ of changing the value of v from d to d' is simply the summed costs of the

actions corresponding to a shortest path from d to d' in the domain transition graph of v. In this case, the causal graph heuristic and the additive heuristic are equivalent and compute an optimal value for $c(v, s(v), s_\star(v)) = h^{\text{add}}(v \mapsto s_\star(v)|s)$ (Helmert and Geffner, 2008).

For any variable v that is not a root node, changing the value of v via an action a may require (due to the preconditions of a) that some predecessors of v in the causal graph have certain values at the time of applying a. In this case, both the additive heuristic and the causal graph heuristic recursively calculate the costs of making the necessary changes to the predecessors of v. The difference is that the additive heuristic uses the current state s as the basis for all recursions (see 2.1), whereas the causal graph heuristic keeps track of changes made to predecessor variables in earlier recursions. For example, when calculating the costs of changing v from d_0 to $d_\star$ via an intermediate value d, the recursive computation for changing v from d to $d_\star$ is not based on the original state s for which the heuristic is computed, but on a state s' that reflects the changes that were necessary to first change the value of v from d_0 to d.

Calculating the costs $c(v, d, d')$ optimally is intractable (Helmert, 2004), so that the causal graph heuristics greedily commits to (locally optimal) subplans that achieve the preconditions for certain *transitions* of v. (By transition we denote the change between two neighbouring values in the domain transition graph of v.) The combination of the subplans for such individual transitions may however not be an optimal plan for the overall problem of changing v from its start value to its goal value. In fact, the heuristic is incomplete, as the greedy selection of subplans may lead to dead ends. Since the heuristic assumes acyclic causal graphs, it is furthermore necessary to prune cyclic causal graphs before planning (by ignoring some action preconditions in the heuristic), resulting in a loss of heuristic accuracy.

Preferred operators are defined for the causal-graph heuristic as those actions that change the current value $s(v)$ of a goal variable v to a value that is on a lowest-cost path (as computed by the heuristic) to the goal value $s_\star(v)$ in the domain transition graph of v. If no such action exists for v, the process recurses for the variables involved in the preconditions of actions that are necessary to change v (Helmert, 2006).

In more recent work, the causal graph heuristic has been extended to cyclic causal graphs (Helmert and Geffner, 2008). The resulting heuristic, called the *context-enhanced additive heuristic* or h^{cea}, is equivalent to the causal graph heuristic in planning tasks with acyclic causal graphs, but eliminates the need to prune cycles and outperforms the causal graph heuristic empirically. The context-enhanced additive heuristic can be expressed declaratively rather than procedurally, exhibiting its close relationship with the additive heuristic. In particular, if all state variables are of binary range, the context-enhanced additive heuristic is equivalent to the additive heuristic. The details of the context-enhanced additive heuristic are however not central to this thesis, so that we refer to Helmert and Geffner (2008) for its definition.

2.2.3 Search Enhancements

Most successful satisficing planning systems go beyond plain heuristic search by employing various search-enhancement techniques. One example is the use of *helpful actions* (Hoffmann and Nebel, 2001) or *preferred operators* (Helmert, 2006), providing information which may complement heuristic values. A second example is *deferred heuristic evaluation* (Helmert, 2006), a search variant which can reduce the number of costly node evaluations. Another idea is to use more than one heuristic, for example combining the heuristics in a *multi-queue* approach (Helmert, 2006). These techniques have all been used in the Fast Downward planner and have since spread to many newer planners including Temporal Fast Downward (Röger et al., 2008) and systems used for experiments in various publications (Helmert and Geffner, 2008; Keyder and Geffner, 2009). Since the experimental evaluations in this thesis are based on Fast Downward, we will briefly describe the three techniques mentioned above. Chapter 6 contains more detailed descriptions and an experimental study evaluating the benefits of preferred operators and deferred evaluation.

The Multi-Queue Approach for Using Several Heuristics

More than one heuristic can be employed by using a separate open list or *queue* for each heuristic, an approach aimed at exploiting the different strengths of the utilised heuristics in an orthogonal way (Helmert, 2006; Röger and Helmert, 2010). States are always evaluated with respect to all heuristics, and their successors are added to all queues (in each case with the value corresponding to the heuristic of that queue). When choosing which state to expand next, the search algorithm alternates between the different queues, pruning duplicates upon removal. This base scheme can be changed by assigning numerical priorities to each queue and using some queues more often than others.

Preferred Operators

One of the core features of the FF planner is the use of helpful actions (Hoffmann and Nebel, 2001). Helpful actions are actions that are deemed promising by the FF heuristic because they form part of the relaxed plan or achieve one of its preconditions. The idea is that actions that contribute to solving the relaxed task are also more likely than other actions to contribute to solving the original task. FF uses helpful actions to prune the search space and only evaluates successor states reached via helpful actions, an approach that has been shown to improve FF's performance notably (Hoffmann and Nebel, 2001). However, this restriction makes the search incomplete even if used within a complete search algorithm. FF thus does not use this pruning technique in its restarted search if the first, restricted search fails.

In the spirit of FF's helpful actions, Helmert (2006) subsequently coined the term *preferred operators* for all actions that are deemed promising for some reason. We will in the following use that general term to subsume helpful actions. As in the case of FF, these actions are typically derived from the computation of the heuristic value for a state. The Fast Downward planner makes

use of preferred operators with its multi-queue technique, maintaining an additional *preferred-operator queue* for each heuristic. When a state is expanded, those successor states that are reached via preferred operators (the *preferred states*) are put into the preferred-operator queues, in addition to being put into the regular queues like non-preferred states. (Analogously to regular states, any state preferred by at least one heuristic is added to *all* preferred-operator queues. This allows for cross-fertilisation through information exchange between the different heuristics.) States in the preferred-operator queues are thus expanded earlier on average, as they form part of more queues and have a higher chance of being selected at any point in time than the non-preferred states. In particular, at least every second state selected for expansion is a preferred successor. Note that if preferred successors were put *exclusively* into the preferred-operator queues and not into the regular queues as well, the search would expand more non-preferred successors than preferred successors in cases where the fraction of preferred successors is more than 50%. In addition, Fast Downward gives even higher precedence to preferred successors by using the preferred-operator queues more often than regular queues. We describe the details of this mechanism in Chapter 6. In contrast to FF which evaluates *only* preferred successors (if possible), Fast Downward may gain less leverage from its preferred operators, but at the advantage that its search is complete.

Deferred Evaluation

The textbook versions of best-first search as discussed above keep an open list of states to be expanded, ordered according to the heuristic estimates of their goal distance (cost to a goal). In each step, a state with smallest associated value is removed from the open list and expanded by applying all actions applicable in the state. All successor states generated in this way are then evaluated and inserted into the open list. The Fast Downward planner incorporates a variation of best-first search called *deferred heuristic evaluation*, where the successors of a state are not evaluated upon generation. Instead, they are inserted into the open list with the heuristic estimate of their parent. Only upon being removed from the open list for expansion are they evaluated, so that their heuristic value can in turn be used for their successors. According to Helmert (2006), this technique can decrease the number of state evaluations substantially, especially when combined with the use of preferred operators. For instance, consider the way preferred operators are used within Fast Downward (see above). If s' is a preferred successor of a state s, then s' will be expanded sooner than most of its siblings (since the fraction of preferred successors is usually small). If there exists a path with non-increasing heuristic values from s' to the goal, then the remaining siblings of s' will never be evaluated. By contrast, standard best-first search would evaluate all successors of s.

While deferred evaluation may reduce the number of heuristic *evaluations* compared to standard best-first search, it usually *increases* the number of state *expansions*. This is due to the heuristic values being less informative, since they stem from the parent of a state rather than the state itself. However, evaluations are typically costly (e. g., they make up 80% of the runtime in Fast Downward), so that in terms of time, trading evaluations for expansions may be beneficial.

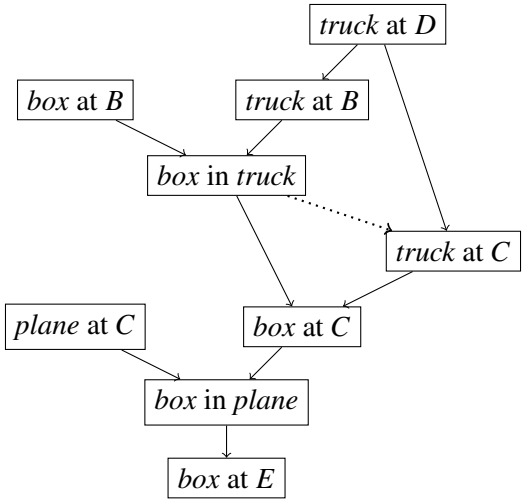


Figure 2.4: Partial landmark graph for the example task shown in Figure 1.1. Bold arcs represent natural orderings, dashed arcs represent reasonable orderings (see Definitions 2.7 and 2.8).

Addressing space, note that successor states need not be generated in an explicit form upon expansion of their parent. It is sufficient to store pointers to the parent state and creating action in the open lists and only explicitly generate a state immediately before its expansion, thus reducing memory requirements. More details on the combination of multi-heuristic search, preferred operators and deferred evaluation in Fast Downward is given in Chapter 6.

2.3 Landmarks

Landmarks are subgoals that must be achieved in every plan (Hoffmann et al., 2004). Using landmarks to guide the search for a plan is an approach that is intuitive for humans. Consider our running example task from Figure 1.1. Arguably the first mental step a human would perform when trying to solve the task is to realise that the box must be transported between two cities, from the left city (locations A–D) to the right city (location E), and that therefore, *the box will have to be transported in the plane*. This in turn means that *the box will have to be at the airport location C*, so that it can be loaded into the plane. This partitions the task into two subproblems, one of transporting the box to the airport location C, and one of delivering it from there to the other city. Both subproblems are smaller and easier to solve than the original task.

Landmarks capture precisely these intermediate conditions that can be used to direct search: the facts L_1 = "*box* is at *C*" and L_2 = "*box* is in *plane*" are landmarks in the task in Figure 1.1. This knowledge, as well as the knowledge that L_1 must become true before L_2, can be automatically extracted from the task in a preprocessing step (Hoffmann et al., 2004). Landmarks and the corresponding ordering knowledge may be represented using a directed graph called the *landmark graph*. A partial landmark graph for our running example is depicted in Figure 2.4.

Landmarks were first introduced by Porteous et al. (2001) and were later studied in more depth by the same authors (Hoffmann et al., 2004). Hoffmann et al. (2004) define landmarks as facts that

are true at some point in every plan for a given planning task. They also introduce *disjunctive landmarks*, defined as sets of facts of which at least one needs to be true at some point. We use a more general definition based on propositional formulas, thus covering both those cases as well as more complex types of landmarks that we will use in Chapter 9. In the following, we refer to landmarks that are facts as *fact landmarks*, and to disjunctions or conjunctions of facts as *disjunctive landmarks* or *conjunctive landmarks*, respectively. Hoffmann et al. show that it is PSPACE-hard to determine whether a given fact is a landmark, and whether an ordering holds between two landmarks. Their complexity results carry over in a straight-forward way to the more general case of propositional formulas.

Definition 2.5. ***Landmarks***
Let $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \rangle$ *be an SAS*$^+$ *planning task, let* $\pi = \langle a_1, \ldots, a_n \rangle$ *be an action sequence applicable in* s_0*, and let* $i, j \in \{0, \ldots, n\}$.

- *A propositional formula* φ *over the facts of* Π *is called a* ***fact formula****.*
- *A fact* F ***is true at time*** i *in* π *iff* $F \in s_0[\langle a_1, \ldots, a_i \rangle]$.
- *A fact formula* φ ***is true at time*** i *in* π *iff* φ *holds given the truth value of all facts of* Π *at time* i*. At any time* $i < 0$*,* φ *is not considered true.*
- *A fact formula* φ *is a* ***landmark*** *of* Π *iff in each plan for* Π*,* φ *is true at some time.*
- *A fact formula* φ ***is added at time*** i *in* π *iff* φ *is true at time* i *in* π*, but not at time* $i - 1$ *(it is considered added at time* 0 *if it is true in* s_0*).*
- *A fact formula* φ ***is first added at time*** i *in* π *iff* φ *is true at time* i *in* π*, but not at any time* $j < i$.

Note that facts in the initial state and facts in the goal are always landmarks by definition.

Landmarks can be classified as causal or non-causal, where the criterion of *causality* excludes "incidentally" achieved landmarks that are added, but not necessarily needed as preconditions by any action in the plan:

Definition 2.6. ***Causal Landmarks***
A fact f *is a* causal (fact) landmark *for a problem* Π *if it is a goal of* Π *or if for all plans* π *for* Π*,* $f \in pre$ *for some* $a = \langle pre, eff \rangle \in \pi$.

The term *action landmark* is used to denote an action that must be part of any plan for a given task (Zhu and Givan, 2003). However, when referring to "landmarks" in this thesis we will not consider action landmarks unless explicitly stated.

Various kinds of *orderings* between landmarks can be defined and exploited during the planning phase. We define three types of orderings for landmarks, which are equivalent formulations of the definitions by Hoffmann et al. (2004), adapted to the SAS$^+$ setting:

Definition 2.7. ***Orderings between landmarks***
Let φ and ψ be landmarks in an SAS$^+$ planning task Π.

- *We say that there is a **natural ordering** between φ and ψ, written $\varphi \rightarrow \psi$, if in each plan where ψ is true at time i, φ is true at some time $j < i$.*
- *We say that there is a **necessary ordering** between φ and ψ, written $\varphi \rightarrow_{\mathrm{n}} \psi$, if in each plan where ψ is added at time i, φ is true at time $i - 1$.*
- *We say that there is a **greedy-necessary ordering** between φ and ψ, written $\varphi \rightarrow_{\mathrm{gn}} \psi$, if in each plan where ψ is* first added *at time i, φ is true at time $i - 1$.*

Natural orderings are the most general; every necessary or greedy-necessary ordering is natural, but not vice versa. Similarly, every necessary ordering is greedy-necessary, but not vice versa. Knowing that a natural ordering is also necessary or greedy-necessary allows deducing additional information, e. g. about plausible temporal relationships between landmarks, as we will describe later on. As a theoretical concept, necessary orderings (φ is *always* true in the step before ψ) are more straightforward and appealing than greedy-necessary orderings (φ is true in the step before ψ becomes true *for the first time*). However, methods that find landmarks in conjunction with orderings can often find many more landmarks when using the more general concept of greedy-necessary orderings (Hoffmann et al., 2004). In our example in Figure 1.1, "*box* is at *B*" is true before "*box* is in *truck*", and the latter must be true before "*box* is at *E*". The first of these orderings is greedy-necessary, but not necessary, and the second is neither greedy-necessary nor necessary, but natural.

Hoffmann et al. (2004) propose further kinds of orderings between landmarks that can be usefully exploited. For example, *reasonable orderings*, which were first introduced in the context of top-level goals (Koehler and Hoffmann, 2000), are orderings that do not necessarily hold in a given planning task. However, adhering to these orderings may save effort when solving the task. In our example task, it is "reasonable" to load the box onto the truck before driving the truck to the airport at *C*. However, this order is not guaranteed to hold in every plan, as it is *possible*, though not "reasonable", to drive the truck to *C* first, then drive to *B* and collect the box, and then return to *C*. The idea is that if a landmark ψ must become false in order to achieve a landmark φ, but ψ is needed after φ, then it is reasonable to achieve φ before ψ (as otherwise, we would have to achieve ψ twice). However, it may actually be necessary to achieve ψ twice, once before and once after φ, in which case the reasonable ordering is not a correct ordering as any plan for the task must violate it.

The idea may be applied iteratively, as we are sometimes able to find new, induced reasonable orderings if we restrict our focus to plans that obey a first set of reasonable orderings. Hoffmann et al. call the reasonable orderings found in such a second pass *obedient-reasonable orderings*. The authors note that conducting more than two iterations of this process was not worthwhile in their

experiments, as it did not result in any notable increase of planning performance. The following definition characterises these two types of orderings formally.

Definition 2.8. ***Reasonable orderings between landmarks***
Let φ and ψ be landmarks in an SAS$^+$ planning task Π.

- *We say that there is a **reasonable ordering** between φ and ψ, written $\varphi \rightarrow_\mathrm{r} \psi$, if for every plan π where ψ is added at time i and φ is first added at time j with $i < j$, it holds that ψ is not true at some time m with $m \in \{i+1, \ldots, j\}$ and ψ is true at some time k with $j \leq k$.*

- *We say that a plan π **obeys** a set of orderings O, if for all orderings $\varphi \rightarrow_\mathrm{x} \psi \in O$, regardless of their type, it holds that φ is first added at time i in π and ψ is not true at any time $j \leq i$.*

- *We say that there is an **obedient-reasonable ordering** between φ and ψ with regard to a set of orderings O, written $\varphi \rightarrow_\mathrm{r}^O \psi$, if for every plan π obeying O where ψ is added at time i and φ is first added at time j with $i < j$, it holds that ψ is not true at some time m with $m \in \{i+1, \ldots, j\}$ and ψ is true at some time k with $j \leq k$.*

A problem with reasonable and obedient-reasonable orderings is that they may be cyclic, i. e., chains of orderings $\varphi \rightarrow_\mathrm{r} \psi \rightarrow_\mathrm{x} \ldots \rightarrow_\mathrm{r} \varphi$ for landmarks φ and ψ may exist (Hoffmann et al., 2004). This is not the case for natural orderings, as their definition implies that they cannot be cyclic in solvable tasks.

In addition, the definitions as given above are problematic in special cases. Note that the definition of a reasonable ordering $\varphi \rightarrow_\mathrm{r} \psi$ includes the case where there exist no $i < j$ such that ψ is added at time i and φ is first added at time j, i. e., the case where it holds that in all plans φ is first added (a) before or (b) at the same time as ψ.[1] While (a) implies that reasonable orderings are a generalisation of natural orderings, which might be regarded as a desirable property, (b) may lead to undesirable orderings. For example, it holds that $\varphi \rightarrow_\mathrm{r} \psi$ and $\psi \rightarrow_\mathrm{r} \varphi$ for all pairs φ, ψ that are first added at the same time in all plans, for instance if φ and ψ are both true in the initial state. Similarly, it holds that $\varphi \rightarrow_\mathrm{r} \varphi$ for all φ. We use these definitions despite their weaknesses here to be consistent with the previous work by Hoffmann et al. Closely connected is the question whether reasonable orderings should be interpreted as strict orderings, where φ should be achieved before ψ (as in the definition of obedience above), or whether we allow achieving φ and ψ simultaneously. We use the strict sense of obedience in order to be consistent with the previous work and because it aligns better with our intended meaning of reasonable orderings, even though this strict interpretation of obedience does not fit the contentious cases discussed above.

[1] According to personal communication with the authors, this case was overlooked by Hoffmann et al.

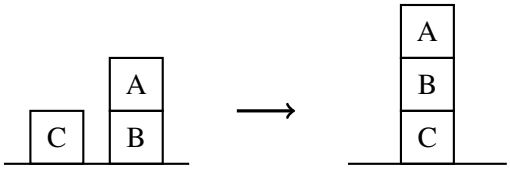


Figure 2.5: A Blocksworld task.

2.4 Benchmark Domains and Performance Criteria

The empirical performance of planning systems is typically evaluated with respect to how many planning tasks from a given benchmark suite they are able to solve within a certain time limit, as well as how efficiently they can solve the tasks (measuring e. g. runtime or the number of expanded search nodes), and how good the solutions are. Between 1998 and 2008, the planning research community has held a series of biennial events where planning systems have been tested empirically against each other: the International Planning Competitions (IPCs). For each such competition, the organisers have established a benchmark set of planning tasks by devising a number of *domains* and a set of tasks for each domain. One example of a planning domain is the Logistics domain from our running example, where objects have to be delivered between locations using various vehicles. Tasks in this domain might differ in the number of objects and vehicles, in the number of cities and in the initial and goal locations of the objects and vehicles. The tasks from the competitions form a standard benchmark set that is widely used for experimental evaluations in the planning literature and will be used in this thesis.

More specifically, the experimental evaluations in this thesis use the tasks of the classical satisficing tracks for fully automated planners of the IPCs 1998–2008. The tasks of 2008 are different from the others in that they specify (non-unit) action costs. In those parts of this thesis that specifically use action costs, we thus evaluate on the tasks from IPC 2008, whereas other parts focus on classical unit-cost planning and use the tasks from the IPCs of 1998 to 2006. (It would be possible to ignore the action costs in the tasks from 2008 and thus use them for our evaluations in classical planning. However, we do not do this, as the tasks from 1998 to 2006 already provide a sufficiently large benchmark set.)

While we concentrate exclusively on the standard IPC benchmarks in our experiments, it is important to note that many other types of planning tasks exist and that the results of any experimental evaluation are highly dependent on the set of benchmarks used. For example, the cost-based planning technique we propose in this thesis does not lead to any benefit on the set of benchmarks tasks from IPC 2008 that we use. By contrast, our approach has been shown to be very useful in tasks derived from diagnosis problems (see the discussion of related work in Section 10.2).

Apart from Logistics, IPC domains that will serve as illustrating examples in this work include *Blocksworld* and *Gripper*, which we briefly explain below. Subsequently, we detail the performance criterion used at IPC 2008 that will also be used in this thesis.

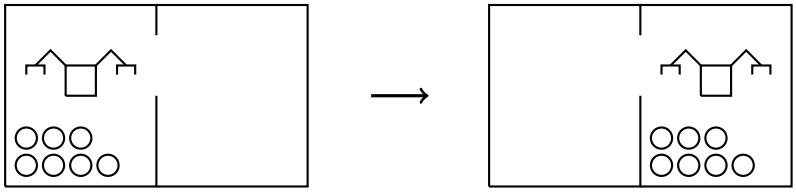


Figure 2.6: A Gripper task.

Blocksworld. Tasks in the Blocksworld domain (IPC 2000) consist of rearranging towers of blocks on a table using a robotic arm. The arm may only hold one block at a time. An example task is depicted in Figure 2.5. Expressed in PDDL, the state variables describe whether the arm is empty, and if not, which block it is holding; and for each block whether it is (a) lying on the table, (b) held by the robotic arm, or (c) stacked on another block, and whether the block is "clear" or has another block stacked on top of it. The actions are to use the arm for picking up or dropping a block from or onto the table, respectively; stacking one block on top of another; or "unstacking" a block from another (i. e., picking up a block that was stacked on another block).

Gripper. Tasks in the Gripper domain (IPC 1998) consist of transporting balls from one room to a second room with the help of a two-armed robot. Each arm of the robot may hold only one ball at a time. An example task is shown in Figure 2.6. State variables describe which room the robot is in, and for each ball either which arm of the robot is holding it (if the ball is being carried) or which room it is lying in (if it is not being carried). Actions are for the robot to pick up a ball or drop a ball, either with its left or right arm, and to move between the two rooms. All balls are typically located in the same room initially and need to be carried to the other room.

The IPC 2008 Performance Criterion. We will measure performance via a number of criteria in this thesis, including coverage (the number of tasks solved from a given set) and plan quality. In some cases it is furthermore desirable to use a ranking criterion that *weighs* the various performance measures of interest against each other, allowing us to rank several planning systems according to one measure. In particular, such an integrated performance criterion is useful when no planner dominates the others consistently (e. g., if one planner solves the most tasks while another planner produces the best plans). One integrated criterion is the *performance score* used at IPC 2008. It combines coverage and plan quality by counting each unsolved task as 0, while solved tasks count between 0 and 1 depending on the quality of the plan. These numbers are then totalled for each planner over all benchmark tasks, and planners are ranked according to their total score. In detail, a planner scores c^*/c for a solved task, where c is the cost of the plan found by the planner and c^* is the cost of the best known solution (e. g., a reference solution calculated by the competition organisers, or the best solution found by any of the participating planners). The influence of coverage on the score depends on whether or not good reference results are available, as we describe below. Nevertheless, the IPC 2008 performance score is a useful criterion for ranking planners if we are interested in tallying both coverage and plan quality with one measure.

For most tasks used at IPC 2008, reference results were generated manually or with domain-specific solvers by the competition organisers, and we will use these reference results in our experimental evaluations. For tasks from previous IPCs no such reference results exist, so that we will take the "best known solution" for these tasks to be the best solution delivered by any of the planners we compare in a given experimental evaluation. This means that the planner producing the best plan for a task is awarded the highest-possible score of 1 in that experiment, even though better plans for that task might exist. This typically increases the influence of *coverage* on the IPC score, since scores for solving a task are generally higher, and thus failure to solve a task has worse relative impact if no reference plans exist. Between several planners that all solve a given task, the lack of a reference plan may slightly skew results in favour of the planner that delivers the best plan, exaggerating the differences between the planners. This effect is however smaller than the impact of coverage described above.

3

Finding Fact Landmarks and Disjunctive Landmarks

In this chapter we discuss methods for discovering landmarks. Section 3.1 describes previous landmark-detection algorithms that work by either backchaining from the goals of the task or propagating information forwards in a relaxed planning graph. In Section 3.2, we extend the previous backchaining approaches and combine them with new techniques for finding and ordering landmarks. Our approach finds disjunctive landmarks, detects additional fact landmarks via the domain transition graphs of a planning task, and infers additional orderings. Section 3.3 contrasts the numbers of landmarks found by our approach and the previous methods. This builds the basis for the following chapter, where we introduce a method of *using* landmarks during planning, and examine how the performance of our approach varies with the set of landmarks found.

3.1 Previous Landmark-Detection Methods

As mentioned previously, deciding whether a given formula is a landmark and deciding orderings between landmarks are PSPACE-hard problems. Thus, practical methods for finding landmarks are incomplete (they may fail to find a given landmark or ordering) or unsound (they may falsely declare a formula to be a landmark, or determine a false ordering). Two types of approaches have been proposed that find fact landmarks and disjunctive landmarks in polynomial time. We briefly discuss these approaches below.

3.1.1 Backchaining from the Goals

Hoffmann, Porteous, and Sebastia (2004) propose an algorithm that extracts fact landmarks and their orderings from the relaxed planning graph for a given task. We denote this algorithm by LM^{HPS} in the following. LM^{HPS} proceeds in three stages. First, potential landmarks and orderings are suggested by a fast *candidate generation procedure*. Second, a *filtering procedure* evaluates a sufficient condition for landmarks on each candidate fact, removing those which fail the test (though unsound orderings may remain). Third, reasonable and obedient-reasonable orderings between the landmarks are derived. Here, we give a high-level overview of the first two stages, adapted to SAS^+ planning (LM^{HPS} is based on the STRIPS formalism). The third stage is ex-

plained in Section 3.1.3.

In general, landmarks can be generated from a set of known landmarks (e. g. the facts in the goal) through backchaining. For example, if a fact B is a landmark that is not already true in the initial state and all achievers of B (actions that have B as an effect) share a common precondition A, then A is a landmark, too: since one of the achievers of B must occur in every plan, any precondition for all achievers – in the following called a *shared precondition* – must be reached by every plan. Unfortunately, with this strong condition few landmarks are usually found (Hoffmann et al., 2004). Instead, we may restrict our attention to those actions which achieve *B for the first time*. These so-called *first achievers* are actions that make B true and can be applied at the end of a partial plan that has never made B true before. However, it is PSPACE-hard to determine the first achievers of a landmark exactly (Hoffmann et al., 2004), so the candidate generation procedure of LM^{HPS} uses an approximation based on the relaxed planning graph.

LM^{HPS} considers those actions to be first achievers of a landmark B that achieve B and first appear in the relaxed planning graph one time step before B. However, it is possible that in a given plan, B is first made true by an action that appears at the same time or after B in the relaxed planning graph. This approximation may thus not capture all first achievers. In detail, the method works as follows. For $i > 0$, let the set O'_i contain precisely the actions which are applicable in the relaxation after i steps, but not previously (i. e., the actions which are part of layer O_i but not O_{i-1} in the relaxed planning graph, see Section 2.2.2). Landmark candidates are then suggested as follows: starting from a known landmark B which first appears in the relaxed planning graph in fact layer S_i, consider all actions in O'_{i-1} that achieve B. If these actions have a common precondition A, then A is a landmark candidate which is ordered *greedy-necessarily* before B.

In addition, LM^{HPS} finds further landmarks with a *one-step look-ahead*: it may happen that the first achievers of a landmark B do not share a precondition, but that there is a fact A which is in turn needed for the preconditions of the first achievers. Let the set of actions $\{a_1, \ldots, a_n\}$ be the set of first achievers of a landmark B, and let $X := \{L_1, \ldots, L_n\}$ be facts s. t. $L_i \in pre(a_i)$ – i. e., each L_i is part of the precondition for a_i. Then X is a disjunctive landmark since one of the facts in X needs to occur in every plan. While LM^{HPS} does not record such disjunctive landmarks, they are used as intermediaries for finding fact landmarks: if the union of all first achievers of the facts in X share a precondition A, then A is a landmark that must occur (at least) two steps before B. To avoid having to test an exponential number of such intermediaries, LM^{HPS} only considers sets X where all facts share the same predicate symbol in the original PDDL representation (see Section 2.1 for a brief description of PDDL).

Due to the approximation of first achievers within the candidate generation procedure, there is no guarantee that the generated candidates are landmarks. Therefore, the filtering procedure in the second stage applies a sufficient criterion to eliminate non-landmarks. Each fact A is tested by removing all achievers of A from the original task and then checking whether the resulting task still has a relaxed solution. If not, then A is indeed critical to the solution of the original task and is thus guaranteed to be a landmark. Otherwise, the candidate A is rejected. We remark that

this pruning criterion guarantees that LM^{HPS} only generates true landmarks; however, landmark *orderings* are not pruned, and there is no soundness guarantee for them.

3.1.2 The Possibly-Before Criterion

Porteous and Cresswell (2002) propose a different approximation for the set of *first achievers* of B that considers more actions and guarantees the correctness of the found landmarks and orderings. Rather than building the relaxed planning graph using all actions and stopping when B first occurs, any achiever of B is left out during the construction of the graph. The fixpoint layer of this *restricted relaxed planning graph* represents an over-approximation of the set of facts that can be achieved *before* B in the planning task; we denote it by $pb(B)$ (for *possibly before*). Any action that achieves B and is applicable given $pb(B)$ qualifies as being *possibly applicable before* B in the original task and is called a *possible first achiever* in the following. By taking the intersection over the shared preconditions of these possible first achievers, the approach may consider too many achievers and miss out on some landmarks. However, since any action that is indeed applicable before B in the original task will be contained in the set of possible first achievers, this approach guarantees the correctness of the found landmarks. This means there is no need for a subsequent filtering procedure as in LM^{HPS} and no unsound orderings will be generated. Porteous and Cresswell furthermore retain the disjunctive landmarks found by the backchaining procedure rather than using them only as intermediaries like LM^{HPS} does.

3.1.3 Finding Reasonable and Obedient-Reasonable Orderings

When the backchaining method for landmark discovery has finished, Hoffmann et al. (2004) introduce *reasonable* and *obedient reasonable* orderings (see Definition 2.8) between the found landmarks. These types of orderings are not always sound, but improve planning performance in practice (Hoffmann et al., 2004).

There is a reasonable ordering $L \rightarrow_{\text{r}} L'$ between two (distinct) fact landmarks L and L' if it holds that (a) L' must be true at the same time or after first achieving L, and (b) achieving L' before L would require making L' false again to achieve L. Hoffmann et al. (2004) approximate both (a) and (b) with sufficient conditions. In the case of (a), they test if $L' \in s_\star$ or if the preceding backchaining method has found a chain of greedy-necessary orderings between landmarks $L = L_1 \rightarrow_{\text{gn}} \ldots \rightarrow_{\text{gn}} L_n$, with $n > 1$, $L_{n-1} \neq L'$ and a greedy-necessary ordering $L' \rightarrow_{\text{gn}} L_n$. (Note that this could be generalised to chains of natural orderings, but all the orderings found by Hoffmann et al. are greedy-necessary.) For (b) they check whether (i) L and L' are inconsistent, i. e. mutually exclusive, or (ii) all actions achieving L have an effect that is inconsistent with L', or (iii) there is a landmark L'' inconsistent with L' with the ordering $L'' \rightarrow_{\text{gn}} L$. To detect inconsistencies between facts they use a sound but incomplete method by Fox and Long (1998).

In a second pass, obedient-reasonable orderings are added. This is done using the same method as above, except that now reasonable orderings are used in addition to greedy-necessary orderings

to derive the fact that a landmark L' must be true after a landmark L. Finally, Hoffmann et al. use a simple greedy algorithm to break possible cycles due to reasonable and obedient-reasonable orderings in the landmark graph. The algorithm first removes obedient-reasonable orderings that contribute to cycles. Then it removes reasonable orderings until no cycles remain.

3.1.4 Forward Propagation in the Relaxed Planning Graph

Zhu and Givan (2003) propose a technique for finding causal fact landmarks by propagating information about "necessary predecessors" forward in a relaxed planning graph. The algorithm works by associating with each action or fact node at every level of the graph a *label* consisting of the set of facts that must be made true (added) in order to reach it. In the first level of the graph, each initial state fact is associated with a label containing only itself. The labels of the nodes appearing at following levels are obtained by combining the labels of the nodes in previous layers in two different ways:

- The label for an action node a at level i is the union of the labels of all its preconditions at level $i - 1$.
- The label for a fact node f at level i is the intersection of the labels of all action nodes adding it at level $i - 1$ (possibly including no-op actions), plus the fact itself.

Intuitively, these rules state that for a fact f to be a landmark for an action a, it is sufficient that f be a landmark for *some* precondition of a, and that for a fact f to be a landmark for another fact f' at a given level, either $f = f'$ or f must be a landmark for *all* action nodes that can achieve f' at that level.

Given these propagation rules, the label associated with a fact or action node at any level i is a *superset* of the set of causal landmarks for this fact or action in Π^+. If the graph construction continues until a fixpoint is reached, i. e. until no further changes occur in the node labels from layer to layer, the landmarks for the goal nodes in the last layer form the complete set of causal landmarks for the relaxed task (Zhu and Givan, 2003). In addition to propagating facts in the labels, actions may be propagated in an analogous manner to derive *action landmarks* for the task.

3.2 Finding Landmarks for SAS$^+$ Planning

We propose an algorithm for finding fact landmarks and disjunctive landmarks that is partly based on the backchaining methods mentioned above, adapting them to the SAS$^+$ setting. In addition, our algorithm exploits the SAS$^+$ representation by using domain transition graphs to find further landmarks. As a result, it generally finds more landmarks and orderings than previous approaches.

All landmarks we discussed earlier for the example task in Figure 1.1 were facts (see the partial landmark graph in Figure 2.4). However, more complex landmarks may be required in larger tasks. Consider an extended version of the example, where the city on the right has two airports,

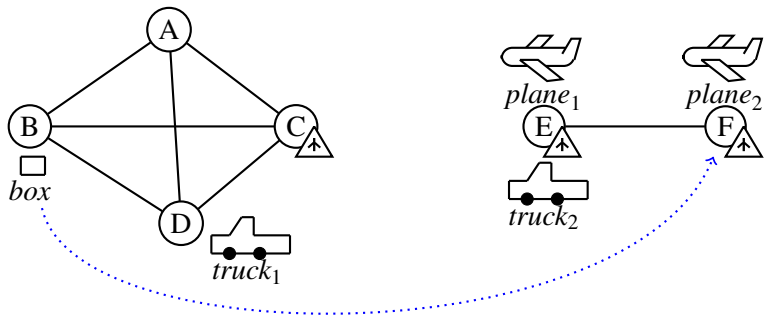


Figure 3.1: An extended Logistics task: transport the box from location B to F.

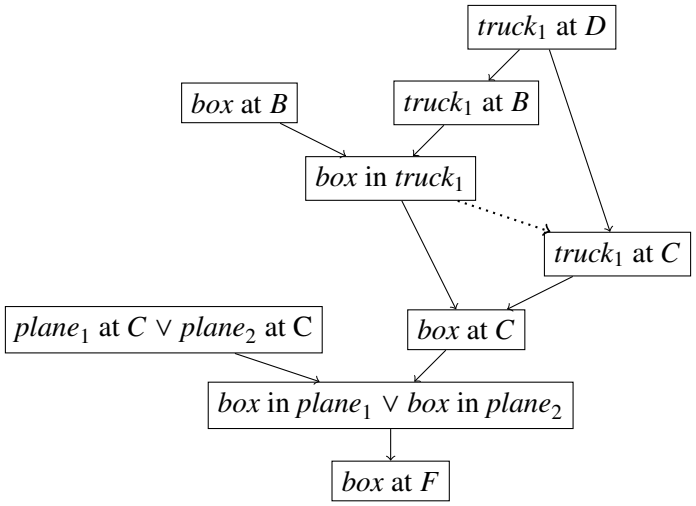


Figure 3.2: Partial landmark graph for the example task shown in Figure 3.1. Bold arcs represent natural orderings, dashed arcs represent reasonable orderings.

and there are multiple planes and trucks, as depicted in Figure 3.1. The previous landmark L_1 = "*box* is at *C*" is still a landmark in our extended example. However, L_2 = "*box* is in *plane*" has no corresponding fact landmark in this task, since neither "*box* is in $plane_1$" nor "*box* is in $plane_2$" is a landmark. The disjunction "*box* is in $plane_1 \vee$ *box* is in $plane_2$", however, is a disjunctive landmark. A partial landmark graph with disjunctive landmarks for our extended example is shown in Figure 3.2. While we will show in the next chapter that the use of disjunctive landmarks improves planning performance, compared to using only fact landmarks, more complex landmarks introduce additional difficulty both with regard to their detection and their handling during planning. The method discussed here is thus only concerned with fact landmarks and disjunctive landmarks, rather than general propositional formulas. Chapter 9 discusses the trade-off between information gain and computational complexity for *conjunctive landmarks*. Fast methods for finding other, more complex types of landmarks are an interesting topic of future work.

Next, we discuss the various components of our method for finding SAS^+ landmarks in detail. High-level pseudo-code for our algorithm, containing the steps described in the following sections 3.2.1–3.2.4, is shown in Algorithm 3.1 on page 36.

3.2.1 Backchaining Using Possible First Achievers

The backchaining part of our algorithm is similar to LM^{HPS} (see Section 3.1.1), but uses the *possibly-before* criterion (see Section 3.1.2) to ensure that only sound orderings are found. Instead of the one-step look-ahead that LM^{HPS} performs to find further landmarks, we opt for the more general approach to admit disjunctive landmarks. Like LM^{HPS} we create disjunctive sets of facts from the preconditions of first achievers of a landmark B such that a set contains one precondition fact from each first achiever of B. Like LM^{HPS}, we require that all facts must stem from the same predicate symbol. Furthermore, we discard any fact sets of size greater than four (though we found this restriction to have little impact compared to the predicate restriction). Each set A found this way is then recorded as a disjunctive landmark and ordered greedy-necessarily before B. If B is a disjunctive landmark, then the first achievers of B are those actions which achieve one of the facts in B.

3.2.2 Landmarks via Domain Transition Graphs

An additional cheap and easy way of extracting more landmarks is offered by the SAS^+ representation using domain transition graphs (DTGs; see Def. 2.3). Given a fact landmark $B = \{v \mapsto l\}$ that is not part of the initial state s_0, consider the DTG of v. The nodes of the DTG correspond to the values that can be assigned to v, and the arcs to the possible transitions between them. If there is a node l' that occurs on *every* path from the *initial state value* $s_0(v)$ to the *landmark value* l, then that node corresponds to a landmark value l' of v: We know that every plan achieving B requires that v takes on the value l', hence the fact $A = \{v \mapsto l'\}$ is a landmark that can be naturally ordered before B. To find these kinds of landmarks, we iteratively remove one node from the DTG and test

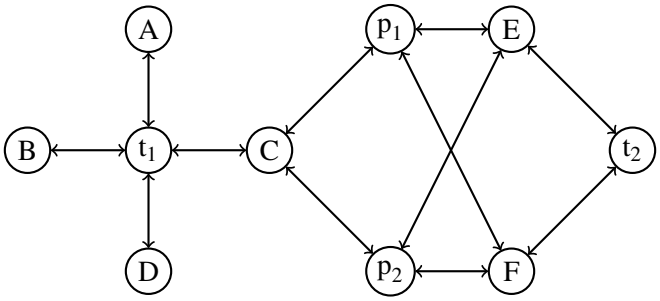


Figure 3.3: Domain transition graph for the location of the box in our extended example (Figure 3.1).

with a simple graph algorithm whether $s_0(v)$ and l are still connected – if not, the removed node corresponds to a landmark. We further improve this procedure by removing, as a preprocessing step, all nodes for which we know that they cannot be true before achieving B, namely the nodes that are not in $pb(B)$ (see Section 3.1.2). Removing these nodes may decrease the number of paths reaching B and may thus allow us to find more landmarks.

Consider again the landmark graph of our extended example, shown in Figure 2.4. Most of its landmarks and orderings can be found via the backchaining procedure described in the previous section, because the landmarks are direct preconditions for achieving their successors in the graph. There are two exceptions: "*box* in $truck_1$" and "*box* at C". These two landmarks are however found with the DTG method. The DTG in Figure 3.3 immediately shows that the box location must take on both the value t_1 and the value C on any path from its initial value B to its goal value F.

3.2.3 Additional Orderings from Relaxed Planning Graphs

For a given landmark ψ, the set $pb(\psi)$ can be used to derive additional orderings. Any landmark χ that is not in this set cannot be reached before ψ, and we can thus introduce a natural ordering $\psi \rightarrow \chi$. Note that the computation of $pb(\psi)$ via the restricted relaxed planning graph of ψ as described in Section 3.1.2 can be easily generalised to disjunctive landmarks. For efficiency reasons, we construct the restricted relaxed planning graph of ψ only once (line 18 in Algorithm 3.1), i. e., when needed to find possible first achievers of ψ during the backchaining procedure. We record potential orderings between ψ and all facts that are not in $pb(\psi)$ (line 30). For all such facts F that are later recognised to be landmarks, we then introduce the ordering $\psi \rightarrow F$ (line 31).

3.2.4 Overlapping Landmarks

Due to the iterative nature of the algorithm it is possible that we find disjunctive landmarks for which at least one of the facts is already known to be a fact landmark. In such cases, we let fact landmarks take precedence over disjunctive ones, i. e., when a disjunctive landmark is discovered that includes an already known fact landmark, we do not add the disjunctive landmark. Conversely, as soon as a fact landmark is found that is part of an already known disjunctive landmark,

Global variables:
$\Pi = \langle \mathcal{V}, s_0, s_\star, O, C \rangle$ ▷ Planning task to solve
$LG = \langle L, O \rangle$ ▷ Landmark graph of Π
queue ▷ Landmarks to be back-chained from

1: **function** ADD_LANDMARK_AND_ORDERING(φ, $\varphi \rightarrow_{\mathrm{x}} \psi$)
2: **if** φ is a fact and $\exists \chi \in L\colon \chi \not\equiv \varphi$ and $\varphi \models \chi$ **then** ▷ Prefer fact landmarks
3: $L \leftarrow L \setminus \{\chi\}$ ▷ Remove disjunctive landmark
4: $O \leftarrow O \setminus \{ (\vartheta \rightarrow_{\mathrm{x}} \chi), (\chi \rightarrow_{\mathrm{x}} \vartheta) \mid \vartheta \in L \}$ ▷ Remove obsolete orderings
5: **if** $\exists \chi \in L\colon \chi \not\equiv \varphi$ and $var(\varphi) \cap var(\chi) \neq \emptyset$ **then** ▷ Abort on overlap
6: **return**
7: **if** $\varphi \notin L$ **then** ▷ Add new landmark to graph
8: $L \leftarrow L \cup \{\varphi\}$
9: $queue \leftarrow queue \cup \{\varphi\}$
10: $O \leftarrow O \cup \{\varphi \rightarrow_{\mathrm{x}} \psi\}$ ▷ Add new ordering to graph

11: **function** IDENTIFY_LANDMARKS
12: $LG \leftarrow \langle s_\star, \emptyset \rangle$ ▷ Landmark graph starts with all goals, no orderings
13: $queue \leftarrow s_\star$
14: $add\_orders \leftarrow \emptyset$ ▷ Additional orderings (see Section 3.2.3)
15: **while** $queue \neq \emptyset$ **do**
16: $\psi \leftarrow$ POP($queue$)
17: **if** $s_0 \not\models \psi$ **then**
18: $RRPG \leftarrow$ the restricted relaxed plan graph for ψ
19: $pre_{shared} \leftarrow$ shared preconditions for ψ extracted from $RRPG$
20: **for** $\varphi \in pre_{shared}$ **do**
21: ADD_LANDMARK_AND_ORDERING(φ, $\varphi \rightarrow_{\mathrm{gn}} \psi$)
22: $pre_{disj} \leftarrow$ disjunctions covering preconditions for ψ given $RRPG$
23: **for** $\varphi \in pre_{disj}$ **do**
24: **if** $s_0 \not\models \varphi$ **then**
25: ADD_LANDMARK_AND_ORDERING(φ, $\varphi \rightarrow_{\mathrm{gn}} \psi$)
26: **if** ψ is a fact **then**
27: $pre_{lookahead} \leftarrow$ find landmarks from DTG of the variable in ψ
28: **for** $\varphi \in pre_{lookahead}$ **do**
29: ADD_LANDMARK_AND_ORDERING(φ, $\varphi \rightarrow \psi$)
30: $add\_orders \leftarrow add\_orders \cup \{\psi \rightarrow F \mid F$ never true in RRPG$\}$
31: add further orderings between landmarks from add_orders

Algorithm 3.1: Identifying landmarks and orderings via backchaining, domain transition graphs and restricted relaxed planning graphs.

Domain	Hoffmann et al.		Zhu & Givan		New detection method (RHW)	
	LMs	Orderings	LMs	Orderings	LMs (Disj./DTG)	Orderings
Airport (50)	**42614**	294965	37156	73850	38203 (1014/7287)	**1459285**
Depot (22)	1420	4937	1240	2629	**1440** (159/179)	**6961**
Freecell (80)	**8448**	38809	7855	13700	7716 (0/2834)	**95330**
Gripper (20)	960	1400	960	1380	**1420** (460/460)	**2780**
Logistics 1998 (35)	2374	5261	2177	1965	**2909** (732/1230)	**8167**
Miconic Simple ADL (150)	6583	8676	**10045**	**11469**	6583 (0/80)	10762
MPrime (35)	**199**	159	132	92	164 (44/51)	**198**
Rovers (40)	**2827**	1946	1565	785	2338 (379/2)	**2095**
Schedule (150)	8572	6508	7555	5491	**11530** (0/2958)	**9466**
Total	121056	433977	**153370**	345515	140630 (4977/19052)	**2104220**

Table 3.1: Numbers of landmarks and orderings found by various landmark-detection methods. For the new method, numbers in parentheses indicate disjunctive landmarks and landmarks found by the domain transition graph method, respectively. Bold results indicate largest number of landmarks/orderings found in a given domain across the three approaches. Totals (last row) are across all domains from the International Planning Competitions 1998–2006 except for the trivial Movie domain.

we discard the disjunctive landmark including its orderings[1] and add the fact landmark instead. To keep the procedure and the resulting landmark graph simple, we furthermore do not allow landmarks to overlap. Whenever some fact from a newly discovered disjunctive landmark is also part of some already known landmark, we do not add the newly discovered landmark. All these cases are handled in the function ADD_LANDMARK_AND_ORDERING (lines 1– 10).

3.2.5 Reasonable and Obedient-Reasonable Orderings

As a post-processing step, we introduce reasonable and obedient reasonable orderings in a very similar way to Hoffmann et al. (see Section 3.1.3). One difference between their approach and ours is that we also make use of the *natural orderings* we find when approximating whether one landmark must be true after (or at the same time) as another. Furthermore, we use a different method for deriving inconsistencies between facts. Inconsistencies can be identified very easily in the SAS$^+$ representation if the facts are of the form $v \mapsto d$ and $v \mapsto d'$, i. e., if they map the same variable to different values. In addition, our implementation based on the Fast Downward planner uses the groups of inconsistent facts that Fast Downward computes while translating PDDL input into a SAS$^+$-based representation (Helmert, 2006, 2009). Thirdly, our method for breaking cycles is slightly different in that it considers each cycle only once, removing obedient-reasonable orderings rather than reasonable orderings in each cycle whenever possible.

3.3 Evaluation and Discussion

We contrast our technique for finding landmarks with the LM$^{\text{HPS}}$ approach by Hoffmann et al. and the forward-propagating method by Zhu and Givan that we denote by LM$^{\text{ZG}}$. Our approach will

[1] Note that an ordering $\{F, G\} \rightarrow \psi$ neither implies $F \rightarrow \psi$ nor $G \rightarrow \psi$ in general. Conversely, $\varphi \rightarrow \{F, G\}$ neither implies $\varphi \rightarrow F$ nor $\varphi \rightarrow G$.

be referred to as LM^{RHW}, or short RHW, in later chapters of this thesis, denoting the authors of the article where these results were first published (Richter, Helmert, and Westphal, 2008).

Table 3.1 shows the number of landmarks and orderings for the three approaches in some example domains, and summed up over all domains of the International Planning Competitions 1998–2006 except for the trivial Movie domain. None of the landmark-finding approaches dominates the others consistently: for each approach, there is a domain where it finds more landmarks than the others. The LM^{ZG} procedure only detects *causal* landmarks and often finds fewer landmarks and orderings than the other two methods; for example, this is the case in the Airport domain. Our detection method finds slightly more landmarks than LM^{HPS} on average. This is mainly due to disjunctive landmarks and landmarks found via domain transition graphs. LM^{HPS} finds more fact landmarks than we do by backchaining, but at the expense of possibly introducing incorrect orderings. When looking at orderings, we find that our approach finds many more orderings than both other approaches.

Note that all methods discussed here find only landmarks that correspond to landmarks in the *delete-relaxation* of the given task. For the backchaining methods this is due to the approximation of first achievers with the relaxed planning graph. Similarly, LM^{ZG} relies on the relaxed planning graph for its landmark detection. When considering domain transition graphs, note that an edge from d to d' exists in the DTG of variable v if there exists an action that could *possibly* change the value of v from d to d', irrespective of whether or not a state can be reached where that action is applicable. Any landmark found via the DTG method thus also corresponds to a landmark in the delete relaxation for the task. This remains true even if our method removes nodes from the DTG, as we only remove nodes that are not even relaxed reachable before the target node. In Chapter 9, we introduce a method for finding landmarks that are not delete-relaxation landmarks.

Finally, some remarks on runtime. Computing landmarks is usually very inexpensive, as relaxed planning graphs can be built in linear time. For most tasks, landmark computation time is below one second on our benchmark set. Therefore, overall runtime is dominated by the subsequent planning time for all but the simplest planning tasks.

4

Using Landmarks as a Heuristic

Having discussed how to find landmarks in the previous chapter, we now look at ways of exploiting them in planning. In Section 4.1, we review existing approaches, with the most successful method using landmarks as subgoals in decomposed planning tasks. In Section 4.2, we propose an approach that uses landmarks to derive a pseudo-heuristic, which can be usefully combined with other heuristics to improve planning performance. Section 4.3 evaluates our method experimentally, showing that it leads to improved coverage and better solution quality, compared to both the previous method of using landmarks as subgoals and to a base planner not using landmarks.

4.1 Previous Methods for Using Landmarks

For exploiting landmarks during search, Hoffmann et al. (2004) propose a procedure that decomposes the planning task into smaller subtasks by making the landmarks intermediary goals. Instead of searching for the goal of the task, this approach iteratively aims to achieve a landmark that is minimal with respect to the orderings. We will denote this procedure by LM^{local}.

In detail, LM^{local} first builds a landmark graph (with landmarks as vertices and orderings as arcs). Possible cycles are broken by removing some arcs. The sources S of the resulting directed acyclic graph are handed over to a base planner as a disjunctive goal, and a plan is generated to achieve one of the landmarks in S. This landmark, along with its incident arcs, is then removed from the landmark graph, and the process repeats from the end state of the generated plan. Once the landmark graph becomes empty, the base planner is asked to generate a plan to the original goal. (Note that even though all goal facts are landmarks and were thus achieved previously, they may have been violated again.)

As a base planner for solving the subtasks any planner can be used; Hoffmann et al. experimented with FF. They found that the decomposition into subtasks can lead to a more directed search, solving larger instances than classical FF in many domains. However, they also observed that solutions were often longer than those produced by classical FF, as the disjunctive search control can frequently switch between different parts of the task which may have destructive interactions. Sometimes this even leads to dead ends, so that LM^{local} fails on solvable tasks.

In an extension to this work, Sebastia et al. (2006) employ a refined preprocessing technique

that groups landmarks into consistent sets minimising the destructive interactions between the sets. Taking these sets as intermediary goals, they avoid the increased plan length. However, the preprocessing is computationally expensive and may take longer than solving the original problem.

Zhu and Givan (2003) use the causal fact landmarks and action landmarks to estimate the goal distance of a given state. To this end, they treat each fact landmark as a *virtual action* (a set of actions that can achieve the fact landmark) and obtain a distance estimate by bin packing. The items to be packed into bins are the real landmark actions (singletons) and virtual actions, where each bin may only contain elements such that a pairwise intersection of the elements is non-empty. Zhu and Givan employ a greedy algorithm to estimate the minimum number of bins and use this value as a distance estimate. Their experimental results are preliminary, however, and do not demonstrate a significant advantage over the FF planner.

4.2 The Landmark Heuristic

Our aim is to incorporate landmark information into a search for the original goal of the planning task. For this purpose, it is desirable to be able to smoothly integrate the landmark information with other useful heuristics.

A straightforward way of using landmark information for search is to approximate the goal distance of a state s by the estimated number of landmarks that still need to be achieved from s onwards. Given a path (i. e., a sequence of states) π to s, these landmarks are given by

$$L(s,\pi) \coloneqq (L \setminus \mathit{Accepted}(s,\pi)) \cup \mathit{ReqAgain}(s,\pi)$$

where L is the set of all discovered landmarks, $\mathit{Accepted}(s,\pi)$ is the set of *accepted* landmarks, and $\mathit{ReqAgain}(s,\pi)$ is the set of accepted landmarks that are *required again*, with the following definitions based on a given landmarks graph (L, O):

$$\begin{aligned}
\mathit{Accepted}(s,\pi) \quad &\coloneqq \quad \begin{cases} \{\, \psi \in L \mid s \models \psi \text{ and } \nexists(\varphi \rightarrow_{\mathrm{x}} \psi) \in O \,\} & \pi = \langle\rangle \\ \mathit{Accepted}(s_0[\pi'],\pi') \cup \{\, \psi \in L \mid s \models \psi & \pi = \pi';\langle o\rangle \\ \quad \text{and } \forall(\varphi \rightarrow_{\mathrm{x}} \psi) \in O : \varphi \in \mathit{Accepted}(s_0[\pi'],\pi') \,\} & \end{cases} \\
\mathit{ReqAgain}(s,\pi) \quad &\coloneqq \quad \{\, \varphi \in \mathit{Accepted}(s,\pi) \mid s \not\models \varphi \\
&\qquad \text{and } (s_\star \models \varphi \text{ or } \exists(\varphi \rightarrow_{\mathrm{gn}} \psi) \in O : \psi \notin \mathit{Accepted}(s,\pi)) \,\}
\end{aligned}$$

A landmark φ is first accepted in a state s if it is true in that state, and all landmarks ordered before φ are accepted in the predecessor state from which s was generated. Once a landmark has been accepted, it remains accepted in all successor states. For the initial state, accepted landmarks are all those that are true in the initial state and do not have any predecessors in the landmark graph. An accepted landmark φ is *required again* if it is not true in s and (a) it forms part of the goal or (b) it must be true directly before some landmark ψ (i. e., $\varphi \rightarrow_{\mathrm{gn}} \psi$) where ψ is not accepted

in s. In the latter case, since we know that ψ must still be achieved and φ must be true in the time step before ψ, it holds that φ must be achieved again. The number $|L(s,\pi)|$ is then the heuristic value assigned to state s. Pseudo-code for the heuristic is given in Algorithm 4.1.

Global variables:
  $\Pi = \langle \mathcal{V}, s_0, s_\star, O, C \rangle$ ▷ Planning task to solve
  $LG = \langle L, O \rangle$ ▷ Landmark graph of Π
  Accepted ▷ Landmarks accepted in states evaluated so far

function LM_COUNT_HEURISTIC(s, π)
  if $\pi = \langle\rangle$ **then** ▷ Initial state
    $Accepted(s,\pi) \leftarrow \{ \psi \in L \mid s_0 \models \psi \text{ and } \nexists(\varphi \rightarrow_{\mathrm{x}} \psi) \in O \}$
  else
    $\pi' \leftarrow \langle o_1, \ldots, o_{n-1} \rangle$ for $\pi = \langle o_1, \ldots, o_n \rangle$
    $parent \leftarrow s_0[\pi']$ ▷ $Accepted(parent, \pi')$ has been calculated before
    $Reached \leftarrow \{ \psi \in L \mid s \models \psi \text{ and } \forall(\varphi \rightarrow_{\mathrm{x}} \psi) \in O \colon \varphi \in Accepted(parent, \pi') \}$
    $Accepted(s,\pi) \leftarrow Accepted(parent, \pi') \cup Reached$
  $NotAccepted \leftarrow L \setminus Accepted(s,\pi)$
  $ReqGoal \leftarrow \{ \varphi \in Accepted(s,\pi) \mid s \not\models \varphi \text{ and } s_\star \models \varphi \}$
  $ReqPrecon \leftarrow \{ \varphi \in Accepted(s,\pi) \mid s \not\models \varphi \text{ and } \exists \psi \colon (\varphi \rightarrow_{\mathrm{gn}} \psi) \in O \wedge \psi \notin Accepted(s,\pi) \}$
  return $|NotAccepted \cup ReqGoal \cup ReqPrecon|$

Algorithm 4.1: The landmark-count heuristic.

The landmark heuristic will assign a non-zero value to any state that is not a goal state, since goals are landmarks that are always counted as *required again* per condition (a) above. However, the heuristic may also assign a non-zero value to a *goal state*. This happens if plans are found that do not obey the reasonable orderings in the landmark graph, in which case a goal state may be reached without all landmarks being accepted.[1] Hence, we need to explicitly test states for the goal condition in order to identify goal states.

Note that $|L(s,\pi)|$ is not a proper state heuristic in the usual sense, as its definition depends on the way s was reached during search. Nevertheless, it can be used like a heuristic in greedy best-first search. For simplicity, we use the term *landmark heuristic* for this landmark "pseudo-heuristic".

Simply using pure landmark counting, as outlined above, in greedy best-first search already leads to good results in some cases, though it is not competitive with established heuristics such as the FF heuristic. The results can be substantially improved by combining landmark counting with other heuristics (via the multi-queue method), and by using *preferred operators*. We take an action to be a preferred operator in a state if applying it achieves an *acceptable* landmark in the next step, i. e., a landmark whose predecessors have already been accepted. If no acceptable

[1] In the special case where $\varphi \rightarrow_{\mathrm{r}} \psi$ and φ and ψ can become true simultaneously, we could avoid this by accepting both φ and ψ at once (Buffet and Hoffmann, 2010), or we could modify our definition of reasonable orderings such that $\varphi \rightarrow_{\mathrm{r}} \psi$ does not hold unless ψ must become true strictly after φ. The general problem that goal states may be assigned a non-zero value, however, still persists even with these modifications.

landmark can be achieved within one step, the preferred operators are those actions which occur in a relaxed plan to the nearest acceptable fact landmark. This nearest landmark can be computed via a relaxed exploration, using a generalised Dijkstra shortest-path algorithm (see Section 2.2.2), and determining the earliest occurrence of an acceptable landmark in this structure. A relaxed plan to this landmark is then extracted, and the actions in this plan form preferred operators if they are applicable in the current state.

In the next section, we evaluate our new technique for using landmarks – a heuristic search using the landmark heuristic in combination with other heuristics and with preferred operators – on standard benchmark tasks.

4.3 Evaluation

We evaluate our proposed method on all planning tasks from the International Planning Competitions 1998–2006 except the trivial Movie domain. Since our approach is not cost-sensitive, the cost-based tasks from IPC 2008 were not used in this evaluation (see Section 2.4). In all experiments, the time and memory limits are 30 minutes and 3 GB respectively for each task, running on a 2.66 GHz Intel Xeon CPU. As a framework, we use the Fast Downward planner that already contains the functionality to combine various heuristics and to use preferred operators (see Section 2.2.3).

Since the *detection method* for landmarks is orthogonal to their *usage* during search, and furthermore search algorithms using landmarks can be combined with different *base planners*, we can vary three independent dimensions. In order to keep the number of configurations manageable, we conduct two different experiments, with one of the three dimensions fixed in each of them.

4.3.1 Comparing Usages for Landmarks

In the first experiment, we compare our new method for using landmarks as a heuristic (*heur*) to a base planner not using landmark information (*base*) and to the local search algorithm by Hoffmann et al. (*local*), where landmarks are used to decompose the task and the base planner is used for each subtask. We perform this comparison with three different heuristics in the base planner, namely the FF/add heuristic, causal graph heuristic, and a "blind" heuristic assigning 1 to non-goal states and 0 to goal states. The base planner conducts a greedy best-first search with preferred operators and deferred evaluation as implemented in Fast Downward (see Algorithm 6.1). In this experiment, we keep the landmark detection method fixed. Specifically, we use the LM^{HPS} algorithm by Hoffmann et al. (2004) (see Chapter 3) for finding landmarks. Preferred operators are used in all applicable cases (i. e., whenever a non-blind heuristic is used), and reasonable orderings are used in all configurations using landmarks.

Table 4.1 shows the percentage of tasks solved by each algorithm in each domain. With all three base planners, the approach using the landmark heuristic outperforms the other two alternatives (*base* and *local*). The results show that the landmark heuristic can be beneficially combined

Domain	FF heuristic			CG heuristic			blind heuristic		
	base	local	heur	base	local	heur	base	local	heur
Airport (50)	**72**	32	64	46	20	**48**	34	10	**64**
Assembly (30)	100	97	100	10	20	**83**	0	0	**7**
Blocks (35)	100	100	100	100	100	100	43	69	**100**
Depot (22)	86	**100**	95	45	18	**100**	9	82	**95**
Driverlog (20)	100	100	100	100	100	100	25	70	**100**
Freecell (80)	95	80	**98**	89	60	**98**	16	66	**98**
Grid (5)	100	100	100	80	100	100	20	80	**100**
Gripper (20)	100	100	100	100	100	100	25	100	100
Logistics 1998 (35)	94	100	100	100	100	100	6	29	**97**
Logistics 2000 (28)	100	100	100	100	100	100	36	100	100
Miconic (150)	100	100	100	100	100	100	27	100	100
Miconic Full ADL (150)	91	91	90	89	**91**	90	41	**43**	42
Miconic Simple ADL (150)	100	100	100	100	100	100	37	100	100
MPrime (35)	89	80	**97**	100	100	100	37	49	**86**
Mystery (30)	53	53	**57**	57	57	**60**	37	43	**53**
Openstacks (30)	100	100	100	70	23	**100**	23	70	**100**
Optical Telegraphs (48)	4	**8**	4	2	0	**6**	2	8	**100**
Pathways (30)	93	**100**	97	23	27	**100**	13	17	**100**
Philosophers (48)	96	67	**100**	100	10	100	8	10	**73**
Pipesworld Notankage (50)	84	78	**88**	48	38	**84**	22	56	**78**
Pipesworld Tankage (50)	78	58	**86**	28	26	**64**	12	30	**66**
PSR Large (50)	64	**66**	64	64	64	62	20	22	**64**
PSR Middle (50)	100	100	100	100	100	100	50	54	**100**
PSR Small (50)	100	100	100	100	100	100	94	96	**100**
Rovers (40)	100	100	100	80	65	**100**	10	43	**100**
Satellite (36)	97	97	97	97	97	97	11	22	**97**
Schedule (150)	99	61	**100**	99	67	**100**	7	25	**94**
Storage (30)	63	63	60	67	**70**	63	40	50	**57**
TPP (30)	100	100	100	77	77	**100**	17	77	**100**
Trucks (30)	40	3	40	30	3	30	13	7	**23**
Zenotravel (20)	100	100	100	100	100	100	35	90	**100**
Averaged over domains	87	82	**88**	74	66	**87**	25	52	**84**

Table 4.1: Percentage of tasks solved using three different base planners (FF/add heuristic, causal graph heuristic, blind heuristic) and three different methods for using landmarks (base planner using no landmarks, Hoffmann et al.'s LM^{local} algorithm, the landmark heuristic). Bold results indicate better performance than the other two methods for a given base planner and domain. In all cases, Hoffmann et al.'s LM^{HPS} algorithm was used for detecting landmarks and orderings. (Total number of tasks in each domain is shown in parentheses after the domain name in all tables.)

Domain	FF heuristic	
	base	heur
Airport (50)	**6**	2
Depot (22)	0	**2**
Freecell (80)	1	**3**
Logistics 1998 (35)	0	**2**
Miconic Full ADL (150)	**2**	0
MPrime (35)	0	**3**
Mystery (30)	0	**1**
Pathways (30)	1	**2**
Philosophers (48)	0	**2**
Pipesworld Notankage (50)	0	**2**
Pipesworld Tankage (50)	1	**5**
Schedule (150)	0	**1**
Storage (30)	**1**	0
Total	12	**25**

Table 4.2: Comparing the number of tasks solved exclusively by the FF/add-heuristic base planner and the landmark heuristic approach, respectively. An entry of *n* for a given approach and domain means that the approach solved *n* tasks in this domain that the other approach did not solve. Domains where both approaches solved the same set of tasks are not shown.

with a base heuristic: when using the FF/add or causal graph heuristic, the results are significantly better than with the blind heuristic. However, *heur* still performs well even with the blind heuristic, demonstrating that the landmark heuristic is powerful in itself. In contrast, the local landmarks search algorithm, when used in conjunction with the FF/add or causal graph heuristic, is worse than the base planner on average. This is mostly due to the incompleteness of the LM^{local} approach that is prone to getting stuck in dead ends.

Overall, the best results are achieved when using our landmark technique in combination with the FF/add heuristic. The average difference of 1 percentage point between the combination of the landmark heuristic with the FF/add heuristic on the one side and the base planner using only the FF/add heuristic on the other (see the last row of the table) may not seem big at first, but we note that in 10 of the 31 domains, using landmarks leads to more problems being solved than in the base planner, while the converse is only true in 3 domains. Over all domains, there are 25 tasks solved by *heur* but not *base* and 12 tasks solved by *base* but not *heur*. A detailed comparison is shown in Table 4.2.

In this experiment, we did not vary the method of detecting landmarks. We have also run our search algorithm *heur* with landmarks from alternative detection methods, namely the one proposed by Zhu and Givan (2003) and our new method for SAS^+ planning introduced in Chapter 3. We do not report detailed results here, as the average results are very similar to the landmark detection method by Hoffmann et al. In particular, the same average coverage is achieved for all configurations using the FF/add heuristic as a base, while average results for other base planners vary by up to one percentage point.

However, the three approaches have slightly different strengths and weaknesses. For example, Table 4.1 (presenting results for the landmark detection method of Hoffmann et al.) shows that in the Airport domain and using the FF/add heuristic in the base planner, the *base* configuration (no landmarks) solves 72% of the tasks, while the *heur* configuration (using the landmark heuristic) only solves 64%. Hence, we do not seem to find useful landmarks in this domain. Using Zhu and Givan's landmarks, however, *heur* solves 80% of the Airport tasks, a considerable improvement over the baseline (see also Figure 4.1). In the Philosophers domain, on the other hand, Table 4.1 shows that *heur* solves all tasks, while using Zhu and Givan's landmarks reduces the success rate to 75%.

4.3.2 Comparing Landmark-Detection Methods

Interestingly, the performance differences between the landmark detection approaches in our experiment cannot be explained purely by the *number* of landmarks and/or orderings found. Consider again Table 3.1 in Chapter 3, which shows the number of landmarks and orderings for the three approaches. Zhu and Givan's LM^{ZG} procedure finds fewer landmarks and orderings than the other two methods in the Airport domain. At the same time, it leads to the best success rate of the three approaches in this domain.

This is not necessarily due to the fact that LM^{ZG} computes only causal landmarks (as one might assume that non-causal landmarks could be harmful). In Chapter 9, we discuss that even when considering only causal landmarks, it is possible that larger sets of landmarks lead to worse heuristic quality. Furthermore, this holds even in the case where the smaller set of landmarks is a subset of the larger set (unless *optimal cost partitioning* is used, see Chapter 9). Here, we simply note that the sets found by the various detection methods are not necessarily subsets of each other, and that the informativeness of a set of landmarks depends not just on its size, but also on the landmarks found. However, if we assume that none of the detection procedures finds *generally* "better" landmarks than the others, we can typically expect larger sets of landmarks to lead to better planning performance.

Our detection technique finds slightly more landmarks than LM^{HPS} on average, and many more orderings than both other approaches. While this does not affect the average number of tasks solved, it does make a difference in terms of *plan quality*, i. e., the length of the solution plans found.

To highlight this issue, Table 4.3 contains a comparison of plan lengths for various combinations of landmark detection and search procedures.[2] In this second experiment, we always use the FF/add heuristic in the base planner, since this produced the highest success rate in the first experiment. Each column compares the base planner (without landmarks) to a different planner configuration that makes use of landmarks. As expected, the local search algorithm by Hoffmann et al. typically leads to significantly longer plans than the base planner. (For *local*, we only show

[2]The results shown here vary slightly from the ones published earlier (Richter et al., 2008), as we use the more appropriate geometric mean, rather than the arithmetic mean, to compute the average plan length ratios here.

Domain	local-HPS	heur-HPS	heur-ZG	heur-RHW
Airport (50)	1.05	**0.98**	**0.98**	0.99
Assembly (30)	0.99	1.00	**0.98**	1.00
Blocks (35)	1.11	0.87	0.95	**0.82**
Depot (22)	0.79	0.93	**0.78**	**0.78**
Driverlog (20)	0.97	**0.94**	0.95	**0.94**
Freecell (80)	1.11	**1.04**	1.19	1.11
Grid (5)	1.04	1.02	1.03	**0.92**
Gripper (20)	1.07	1.04	1.00	**0.77**
Logistics 1998 (35)	1.05	0.99	1.02	**0.97**
Logistics 2000 (28)	1.13	**1.01**	1.03	**1.01**
Miconic (150)	0.92	0.83	0.98	**0.82**
Miconic Full ADL (150)	1.00	**0.99**	1.06	1.03
Miconic Simple ADL (150)	1.28	1.04	**0.91**	1.03
MPrime (35)	1.00	**0.91**	**0.91**	**0.91**
Mystery (30)	0.94	0.91	0.91	**0.87**
Openstacks (30)	1.03	1.03	**1.02**	**1.02**
Optical Telegraphs (48)	1.00	1.00	1.00	1.00
Pathways (30)	1.01	**0.99**	1.00	1.00
Philosophers (48)	1.24	1.02	**1.01**	1.02
Pipesworld Notankage (50)	1.06	0.96	**0.95**	0.97
Pipesworld Tankage (50)	1.05	0.99	1.09	**0.94**
PSR Large (50)	**0.94**	0.97	0.98	1.00
PSR Middle (50)	**0.94**	0.97	0.99	1.02
PSR Small (50)	1.04	1.01	**1.00**	**1.00**
Rovers (40)	0.98	0.98	0.98	0.98
Satellite (36)	0.93	**0.89**	0.92	0.93
Schedule (150)	0.96	**0.91**	0.95	**0.91**
Storage (30)	1.39	1.16	**1.01**	1.19
TPP (30)	1.31	**0.97**	**0.97**	0.98
Trucks (30)	1.10	**1.00**	1.01	**1.00**
Zenotravel (20)	1.13	1.03	1.03	**0.99**
Averaged over domains	1.04	0.98	0.98	**0.96**

Table 4.3: Plan length comparison. Each result column compares a configuration using landmarks to the base planner. An entry such as "1.05" indicates that the plans found by the landmark approach were 5% longer on average than those of the base planner. Geometric means were used to average over individual instances. The landmarks configurations use different methods for using landmarks (*local*: local landmarks search approach; *heur*: landmark heuristic) and different landmark detection methods (*HPS*: Hoffmann et al.; *ZG*: Zhu & Givan; *RHW*: our method from Chapter 3). All configurations are based on best-first search with the FF/add heuristic and we only compare on instances solved by both approaches.

results for one of the three landmark detection methods. The general observation holds for all three methods.) We found that in 5 (10) domains, *local* increases the plan length for more than 80% (50%) of the tasks, compared to the base planner. In 4 domains, the increase is more than 20% on average (see Table 4.3). In Storage, it is a striking 39%.

By contrast, using the landmark heuristic *reduces* plan lengths compared to the base planner. This is true for all three landmark detection methods; however, the best result is achieved when using our landmarks detection procedure (*heur-RHW* in the table). Compared to the base planner, it decreases plan length by 4% on average, with its best domain being Gripper, where plan length is decreased in every task (by 23% on average), and its worst domain Storage, where plan length is increased in 26% of the tasks (by 19% on average).

The Gripper domain (see Section 2.4) is an example where disjunctive landmarks are particularly helpful. When using only fact landmarks, all landmarks are of the form "at *ball1 roomb*" and "at *robot roomb*". This means that after picking up a ball in room *a*, the fastest way to achieve a new landmark is to move the robot to room *b* and drop the ball there. Such a landmark search results in plans where each ball is carried individually. With disjunctive landmarks, we have additional landmarks of the form "carry *ball1 right* $\vee$ carry *ball1 left*". This means that when the robot has picked up one ball, it can immediately achieve a new landmark by picking up another ball with its free gripper. As a result, our landmark detection method leads to optimal plans in the Gripper domain (see also Figure 4.1).

4.3.3 Runtime Results

Using the landmark heuristic during search in addition to a base heuristic results in somewhat larger runtime per state expansion (because every state now needs to be evaluated by two heuristics). On small or medium-size problems, this overhead often translates into somewhat longer overall runtime of the landmarks approach. As problems grow larger, however, the higher goal-directedness of the landmark search often pays off, as fewer states are evaluated compared to the base planner. Thus, for more difficult problems, the runtime of the *heur* approach is often lower than for the base planner. Averaged over all tasks solved by both approaches, the runtime of *heur* is at most 18% higher than that of the base planners. (The increase of 18% occurs when using the FF/add heuristic as base and the landmarks of Hoffmann et al.; using our landmarks instead results in an average *decrease* in runtime of 1%). Figure 4.1 shows detailed results for some domains.

4.4 Conclusion

We showed how landmark information can be used in a heuristic search framework to increase the number of problem instances solved and improve the quality of the solutions. As opposed to the previously published landmark approach by Hoffmann et al. (2004), our algorithm cannot run into dead ends and we generally achieve better solutions. Our approach can easily be combined with other heuristic information, while the earlier approach appears not to benefit significantly

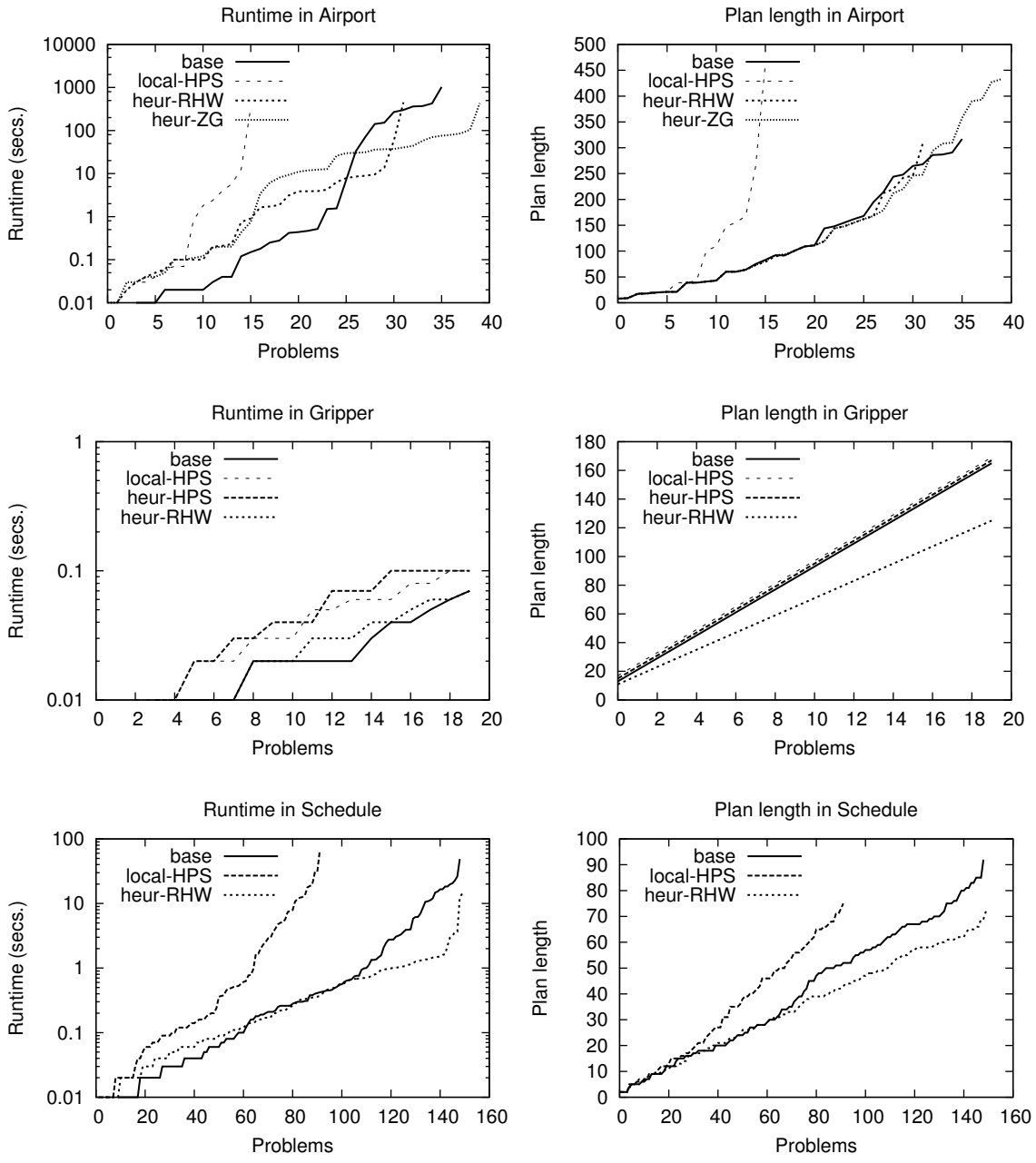


Figure 4.1: Runtimes and plan lengths of various landmark approaches in three planning domains (Airport, Gripper, Schedule). Base planner for all plots is best-first search with the FF/add heuristic. In the graphs on the left, a point at $(10, 0.02)$ indicates that 10 instances from the domain were solved by the respective approach in 0.02 seconds or less. Similarly, in the graphs on the right, a point at $(18, 101)$ indicates that for 18 instances from the domain, solutions of length at most 101 were found.

from additional heuristics. As an example, we showed that two state-of-the-art heuristics, the FF/add heuristic and the causal graph heuristic, can both be significantly improved by integrating landmark information.

5

Landmarks in Cost-Sensitive Planning

In Chapter 4 we showed the performance gain that can be obtained by using landmarks in addition to a base heuristic in satisficing unit-cost planning. In this chapter, we apply the best-performing combination of that work (our RHW landmark-detection procedure together with the FF/add heuristic) to cost-based planning. We find that while the *cost-sensitive* FF/add heuristic generally leads to higher-quality solutions than its cost-unaware counterpart, it results in solving far fewer tasks. Using landmarks proves to be particularly helpful in this case, as the landmarks mitigate the poor coverage of the cost-sensitive FF/add heuristic while typically causing little detriment to plan quality. One core focus of this chapter is to analyse some domains in detail, showcasing the reasons for the bad performance of the cost-sensitive FF/add heuristic and the impact of using landmarks. We present an overview of experimental results in Section 5.3, followed by detailed results in Section 5.4 for four of the nine domains used in our experiments.

5.1 Introduction

While the FF heuristic was originally proposed for unit-cost planning, variants of it can easily be adapted to cost-based planning (see Chapter 2). Keyder and Geffner (2008) implemented the cost-sensitive FF/add heuristic ($h^{\text{FF/add}}$) in the FF planner, and compared it to the original system using the cost-unaware FF heuristic, as well as to several cost-sensitive planning systems. They found the FF planner variant using cost-sensitive $h^{\text{FF/add}}$ to outperform the other planners in terms of solution quality on all eleven benchmark domains considered. Of these eleven domains, five were adapted from the *numerical planning* track at the International Planning Competition in 2002, and six domains were invented by the authors. Keyder and Geffner mention that runtimes with the (cost-unaware) FF heuristic (h^{FF}) are 2–10 times faster than with cost-sensitive $h^{\text{FF/add}}$, as the latter may lead to longer plans (it aims to minimise plan cost, rather than plan length), and thus the search may have to explore more search nodes. A second reason the authors give for the time difference is that the per-node computation time of cost-sensitive $h^{\text{FF/add}}$ is higher than that of h^{FF} in their implementation. While Keyder and Geffner do not explicitly discuss coverage, nearly all of the tasks depicted in detail in their work were solved both when employing cost-sensitive $h^{\text{FF/add}}$ and when using h^{FF}.

In this chapter, we report results using the cost-sensitive FF/add heuristic on the benchmark set of IPC 2008, with Fast Downward as the base planner. Our results are remarkably different from those of Keyder and Geffner in that using cost-sensitive $h^{\text{FF/add}}$ leads to low coverage in our experiment. This discrepancy is likely due to the different benchmark set we are using. Firstly, it may be caused partly by the nature of the domains in our experiment. A second plausible reason is that our benchmark set contains harder tasks, where the additional runtime needed by cost-sensitive $h^{\text{FF/add}}$ means that some tasks remain unsolved.

In Section 5.4, we inspect some example domains in detail to formulate informed hypotheses about what aspects of our benchmark domains cause cost-sensitive $h^{\text{FF/add}}$ to perform poorly (i. e., why minimising plan cost is significantly harder than minimising plan length). We also show that the results are significantly improved when using a landmark heuristic in addition to the FF/add heuristic, combining the two heuristics via a multi-queue heuristic search as in Chapter 4.

5.2 Experimental Setup

We conduct an experimental evaluation on the IPC 2008 benchmarks with Fast Downward as the base planner. We concentrate on four configurations of the planner: using the FF/add heuristic with and without costs, and with and without the additional use of a landmark heuristic, where the landmark heuristic uses costs if the FF/add heuristic does. The landmark heuristics by themselves (without the FF/add heuristic) produce substantially worse results, so that we discuss such configurations only briefly.

The landmark heuristic from Chapter 4 was adapted to the cost-sensitive setting by *weighting* landmarks with an estimate of their cost. Rather than estimating *goal distance* by counting the number of landmarks that still need to be achieved from a state, we estimate the *cost-to-go* from a state by the sum of all estimated costs of those landmarks. The cost counted for each landmark is the minimum action cost of any of its first achievers. (More sophisticated methods for computing the costs of landmarks are conceivable and are a possible topic of future work.)

In the cost-sensitive FF/add heuristic, we break ties according to distance estimates, i. e., we prefer states that are deemed closer to a goal by the cost-unaware FF/add heuristic. This tie breaking significantly improves results in the presence of zero-cost actions, where the search might otherwise explore long search paths without getting closer to a goal. We also slightly modified the greedy best-first search in Fast Downward by implementing a tie-breaking mechanism: in order to encourage cost-efficient plans without incurring much overhead, the search breaks ties between equally promising states by preferring those states that are reached by cheaper actions, i. e., taking into account the *last action* on the path to the considered state in the search space. (The cost of the *entire* path could only be used at the expense of increased time or space requirements, so we do not consider this.)

Experiments were run on a cluster of machines with Intel Xeon CPUs of 2.66 GHz clock speed, the same hardware that was used at IPC 2008, allowing us to compare to other planning

Domain	$\mathbf{F_c}$ / F		$\mathbf{FL_c}$ / F		$\mathbf{FL_c}$ / $\mathbf{F_c}$	
	Tasks	**C. Ratio**	**Tasks**	**C. Ratio**	**Tasks**	**C. Ratio**
Cyber Security	28	0.64	30	0.54	28	0.81
Elevators	15	1.16	22	1.09	14	0.89
Openstacks	30	0.83	30	1.64	30	1.98
PARC Printer	16	0.79	23	0.87	15	1.05
Peg Solitaire	30	0.87	30	0.91	30	1.04
Scanalyzer	28	0.94	28	0.91	30	0.98
Sokoban	23	0.94	22	0.93	24	0.98
Transport	21	1.01	26	0.87	22	0.89
Woodworking	28	1.02	30	1.09	28	1.07
Average	219	0.88	241	0.95	221	1.06

Table 5.1: Average ratio of the solution costs for various pairs of configurations on the tasks solved by both of the configurations involved. Geometric means were used for averaging.

systems that participated in the competition. The time and memory limits were set to the same values as in the competition, using a timeout of 30 minutes and a memory limit of 2 GB.

5.3 Overview of Results

In the following, we use the short-hands **F** and **FL** to denote our *cost-unaware* planner configurations where the former uses only the cost-unaware FF/add heuristic and the latter uses the combination of the cost-unaware FF/add heuristic and the cost-unaware landmark heuristic. The corresponding *cost-sensitive* configurations are denoted by $\mathbf{F_c}$ and $\mathbf{FL_c}$, respectively.

5.3.1 The Cost-Sensitive FF/Add Heuristic

Results showing *plan quality* are given in Table 5.1, where we compare various configurations pair-wise in order to maximise the number of commonly solved tasks between them. For each pair, we show the average ratios of solution costs per domain and over all domains on their commonly solved tasks. As expected, the cost-sensitive configuration $\mathbf{F_c}$ finds cheaper plans than the cost-unaware configuration **F** on average. In some domains, however, most notably in Elevators, $\mathbf{F_c}$ has actually *worse* solution quality than **F**.

Table 5.2 shows the comparatively low *coverage* that results from using the cost-sensitive FF/add heuristic. As can be seen, $\mathbf{F_c}$ solves significantly fewer tasks than **F**. This poor performance of cost-sensitive $h^{\text{FF/add}}$ is not limited to our planner. Table 5.2 also shows results for the FF planner and the implementation FF(h_a) by Keyder and Geffner (2008) mentioned above, where FF(h_a) is the FF planner except for using cost-sensitive $h^{\text{FF/add}}$ instead of h^{FF}. Analogously to our results, FF(h_a) solves significantly fewer tasks than the FF planner. (Note that the FF planner fared badly in the Cyber Security domain due to problems with reading very large task descriptions.)

In order to weigh coverage against quality, we show results according to the IPC 2008 perfor-

Domain	**F**	**F_c**	**FL**	**FL_c**	**FF**	**FF(h_a)**
Cyber Security	30	28	30	30	4	23
Elevators	30	15	30	22	30	23
Openstacks	30	30	30	30	30	25
PARC Printer	25	16	24	23	30	16
Peg Solitaire	30	30	30	30	30	29
Scanalyzer	28	30	30	30	30	28
Sokoban	25	25	23	24	27	17
Transport	26	22	29	30	29	23
Woodworking	30	28	28	30	17	29
Total	**254**	224	**254**	249	227	213

Table 5.2: Coverage for various planners and configurations.

Domain	**F**	**F_c**	**FL**	**FL_c**	**FF**	**FF(h_a)**
Cyber Security	20	24	20	28	4	20
Elevators	22	9	23	14	21	9
Openstacks	20	23	13	13	21	8
PARC Printer	20	16	23	21	27	16
Peg Solitaire	20	23	20	22	20	21
Scanalyzer	19	21	22	21	24	24
Sokoban	18	20	18	19	21	15
Transport	18	15	24	24	18	15
Woodworking	22	20	20	20	14	22
Total	180	171	182	**183**	169	150
(Total IPC 2008)	-	-	-	-	(176)	(157)

Table 5.3: IPC scores (rounded to whole numbers) for various planners and configurations. Scores for IPC planners were re-calculated (see text).

Domain	Score		Coverage	
	L	**L$_c$**	**L**	**L$_c$**
Cyber Security	6	7	6	7
Elevators	16	11	29	20
Openstacks	13	13	30	30
PARC Printer	17	17	18	18
Peg Solitaire	20	23	28	30
Scanalyzer	22	22	30	29
Sokoban	11	13	12	14
Transport	24	24	30	30
Woodworking	5	6	6	7
Total	134	**136**	**189**	185

Table 5.4: IPC scores (rounded to whole numbers) and coverage for configurations using only the landmark heuristic, not the FF/add heuristic.

mance criterion described in Section 2.4. Table 5.3 contains the IPC scores that are achieved by our configurations as well as FF and FF(h_a).[1] As can be seen, using the IPC performance criterion it is *not* worthwhile to consider action costs in the FF/add heuristic. The better solution quality of **F$_c$** is not enough to offset its worse coverage compared to **F**.

5.3.2 Adding Landmarks

As can be seen in Table 5.2, the use of landmarks in conjunction with the FF/add heuristic increases coverage for the cost-sensitive search to a similar level as that of the cost-unaware search. Table 5.1 shows that we sometimes pay for this improvement in coverage with plan quality: **F$_c$** finds better solutions than **FL$_c$**, and while **FL$_c$** still delivers better plans than **F**, the improvement over **F** is smaller, on average, than without landmarks. However, the negative impact through landmarks is mainly due to one domain, Openstacks, where the plans found by **FL$_c$** are nearly twice as expensive as those found by **F$_c$**. In all other domains, plan quality is either improved or not significantly reduced by using landmarks. According to the IPC performance criterion that balances coverage and quality (see Table 5.3), using landmarks in addition to cost-sensitive $h^{\text{FF/add}}$ pays off notably, with **FL$_c$** performing better than **F$_c$** and similarly to (in fact, slightly better than) **F**.

Except for in the Openstacks domain, **FL$_c$** thus typically dominates **F$_c$** by improving coverage while keeping similar levels of solution quality. Compared to **F**, **FL$_c$** deteriorates coverage slightly but improves solution quality.

As mentioned in Chapter 4, the landmark heuristic by itself is however not competitive. We confirmed this by testing two configurations that use only the landmark heuristic and not the FF/add heuristic, **L** and **L$_c$** (denoting cost-unaware and cost-sensitive search, respectively). **L**

[1] The plans found by FF and FF(h_a) have been obtained from the competition website (Helmert et al., 2008). However, the scores for those plans depend on the best known solutions for the tasks. The scores we show here thus differ from the ones published at IPC 2008, as we have re-calculated them to reflect new best solutions found in our experiments. To illustrate the magnitude of the change, the original total scores of the IPC planners are shown in parentheses in the last table row.

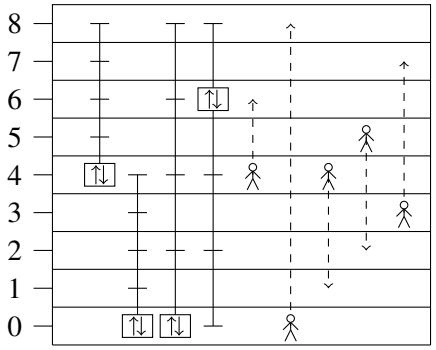


Figure 5.1: An example Elevators task.

and $\mathbf{L_c}$ achieve performance scores of 134 and 136, and coverage points of 189 and 185, respectively (see Table 5.4). This is substantially worse then the scores greater than 171 and coverage points greater than 224 achieved by any of our experimental configurations that use $h^{\text{FF/add}}$. The landmarks-only configurations are however not consistently worse than FF/add-only configurations: in the Scanalyzer and Transport domains, **L** achieves higher coverage and higher performance scores than **F**. However, **L** and $\mathbf{L_c}$ have particularly low coverage in the Cyber Security, Sokoban and Woodworking domains, and low quality in Elevators and Openstacks.

5.4 Detailed Analyses of Select Domains

In the remainder of this chapter, we target the question *why* the cost-sensitive FF/add heuristic leads to such low coverage, and why the corresponding configuration $\mathbf{F_c}$ sometimes even produces plans that are considerably *worse* than those of its cost-unaware counterpart **F**. As we will show in selected domains, the cost-sensitive search often expands many more search nodes than the cost-unaware search, leading to the observed behaviour. This is most likely due to the fact that finding plans of high quality is hard and thus unsuccessful in many of the benchmark tasks. For example, in some domains the cost-sensitive search leads to large local minima that do not exist for the cost-unaware search. More generally, good plans are often longer than bad plans, which may lead to increased complexity in particular in domains where the heuristic values are inaccurate. By way of example, we present detailed results for four of the nine competition domains. These four domains were chosen because they either exaggerate or contradict the average results. The domains Elevators and PARC Printer highlight the problems of the cost-sensitive FF/add heuristic, providing informed hypotheses of why the heuristic performs badly and showing to what extent the use of landmarks overcomes these problems. In Cyber Security, the cost-sensitive heuristic performs uncharacteristically well, while in Openstacks using landmarks does not lead to the usual improvement, but rather to a degradation in performance.

5.4.1 Elevators

The Elevators domain models the transportation of passengers in a building via fast and slow elevators, where each elevator has a certain passenger capacity and can access certain floors. Passengers may have to change elevators to get to their final destination, and furthermore the two different types of elevators have different associated cost functions. This is in contrast to the Miconic domain, used in an earlier International Planning Competition (Bacchus, 2001), which also models the transporting of passengers via elevators, but with a single elevator and unit-cost actions. In Elevators, the floors in the building are grouped into blocks, overlapping by one floor. Slow elevators only operate within a block and can access all floors within their block. Fast elevators can access all blocks, but only certain floors within each block (in the first 10 IPC tasks every second floor, and in the other 20 tasks every fourth floor). Fast elevators are usually more expensive than slow elevators except for a distance of two floors, where both elevator types cost the same. However, fast elevators may sometimes be advantageous when transporting passengers between blocks (as they avoid the need for passengers to switch elevators on the shared floor between blocks), and they usually have a higher capacity.

An example task with eight floors, grouped into two blocks, is shown in Figure 5.1. There are a total of four elevators, two slow ones and two fast ones. The cost function used in the 30 IPC tasks for moving an elevator from its current location to a target floor is $6 + n$ for slow elevators and $2 + 3n$ for fast elevators, where n is the distance travelled (the number of floors between the current location of the elevator and its target). Actions concerning passengers boarding and leaving elevators are free of cost. Assuming this cost function, it is cheaper in this example to transport the passenger located at floor 0 with the two slow elevators (changing at floor 4) than to use a direct fast elevator.

Elevators is one of the domains where the configurations using cost-sensitive $h^{\text{FF/add}}$ ($\mathbf{F_c}$ and $\mathbf{FL_c}$) solve far fewer tasks than their cost-unaware counterparts. Using landmarks increases coverage, but does not solve the problem completely. Furthermore, it is notable that on the problems that the cost-sensitive configurations do solve, their solutions often have *worse quality* than the solutions of the cost-unaware configurations. Table 5.5 illustrates this for the configurations $\mathbf{F}$ and $\mathbf{F_c}$.

While we do not have a full explanation for why the configurations involving cost-sensitive $h^{\text{FF/add}}$ perform so badly in this domain, several factors seem to play a role. Firstly, in attempting to optimise plan costs, the cost-sensitive FF/add heuristic focuses on relatively complex solutions involving mainly slow elevators and many transfers of passengers between elevators, where the relaxed plans are less accurate (i. e., translate less well to actual plans), than in the case of the cost-unaware heuristic. Secondly, the costs associated with the movements of elevators dominate the heuristic values, causing local minima for the cost-sensitive heuristic. Thirdly, the capacity constraints associated with elevators may lead to plateaus and bad-quality plans in particular for the cost-sensitive heuristic. In the following sections, we describe each of these factors in some detail.

	Quality (Score)		Length	
Task	**F**	**F_c**	**F**	**F_c**
01	0.57	**0.59**	26	**24**
02	0.69	**0.72**	27	**25**
03	**0.88**	0.58	**21**	41
04	**0.71**	0.70	**34**	45
05	**0.68**	0.54	**33**	50
06	0.60	0.60	**56**	64
07	0.38	**0.46**	**71**	81
08	**0.84**	0.54	**47**	62
09	**0.71**	0.54	**54**	81
11	**0.66**	0.52	**39**	51
12	**0.70**	0.54	**55**	79
13	**0.58**	0.51	**60**	84
14	0.70	0.70	**81**	101
20	**0.67**	0.47	**132**	173
21	0.70	0.63	84	**83**
Avg.	**0.67**	0.58	**55**	67

Table 5.5: Comparison of plan qualities (measured via the IPC scores) and plan lengths for **F** and $\mathbf{F_c}$ in Elevators. Shown are all tasks solved by both configurations, with bold print indicating the better solution.

	Slow moves	Fast moves	Ratio fast/slow
F	275	45	6.11
$\mathbf{F_c}$	405	21	19.29

Table 5.6: Total elevator moves and ratio of fast to slow moves in the plans found by the **F** and $\mathbf{F_c}$ configurations, on the 15 Elevators instances solved by both configurations.

Lastly, we found that the *deferred heuristic evaluation* technique used in Fast Downward (see Section 2.2.3) did not perform well in this domain. When not using deferred evaluation, the $\mathbf{F_c}$ configuration solves 3 additional tasks (though the quality of solutions remains worse than with the **F** configuration). This partly explains why the FF(h_a) planner by Keyder and Geffner (2008) has a substantially higher coverage than our $\mathbf{F_c}$ configuration in this domain. While the two planners use the same heuristic, they differ in several aspects. Apart from deferred evaluation these aspects include the search algorithm (greedy best-first search vs. enhanced hill-climbing) and the method for using preferred operators (maintaining additional queues for preferred states vs. pruning all non-preferred successor states).

Slow vs. Fast Elevators

When examining the results, we found that the $\mathbf{F_c}$ configuration tends to produce plans where slow elevators are used for most or all of the passengers, while the **F** configuration uses fast elevators more often (see Table 5.6). This is not surprising, as for each individual passenger, travelling from their starting point to their destination tends to be *cheaper* in a slow elevator (unless the distance

is very short), whereas *fewer actions* are typically required when travelling in a fast elevator. The independence assumptions inherent in the FF/add heuristic (see Section 2.2.2) lead to constructing relaxed plans that aim to optimise the transportation of each passenger individually, rather than taking synergy effects into account.

The plans produced by $\mathbf{F_c}$ are also longer, on average, than the plans produced by $\mathbf{F}$ (see Table 5.5), one reason for this being that the predominant use of slow elevators requires passengers to change between elevators more often. As plans become longer and involve more passengers travelling in each of the slow elevators, heuristic estimates may become worse. For example, the relaxed plans extracted for computation of the heuristic are likely to abstract away more details if more passengers travel in the same elevator (e. g., since once a passenger has been picked up from or delivered to a certain location, the elevator may "teleport" back to this location with no extra cost in a relaxed plan to pick up or deliver subsequent passengers). Generally, we found that the relaxed plans for the *initial state* produced by $\mathbf{F_c}$ tend to be similar in length and cost to those produced by $\mathbf{F}$, but the final *solutions* produced by $\mathbf{F_c}$ are worse than those of $\mathbf{F}$. One reason for this is probably that the increased complexity of planning for more passenger change-overs between elevators in combination with worse relaxed plans poses a problem to the cost-sensitive FF/add heuristic.

Local Minima Due to Elevator-Movement Costs

Since action costs model distances, the total cost of a relaxed plan depends on the target floors relative to the current position of an elevator. The action costs of moving the elevator usually dominate the estimates of the cost-sensitive FF/add heuristic. Consider the two example tasks in Figure 5.2, which differ only in the initial state of the elevators. The elevators need to travel to all three floors in a solution plan, but due to abstracted delete effects a relaxed plan for the initial state will only include actions that travel to the two floors other than the starting floor of the elevator (i. e., the elevator can be "teleported" back to its starting floor without cost). In the left task, the relaxed cost of visiting all three floors is lower than in the right task, as the cost in the left task is the sum of going from floor 4 to floor 8, and going from floor 4 to floor 0, resulting in a total cost of 10 + 10 = 20. In the right task, the relaxed cost for visiting all floors is the cost of going from floor 0 to floor 4, and from floor 0 to floor 8, resulting in a total cost of 10 + 14 = 24. In the left task, once the passenger has boarded the elevator on floor 4, *all immediate successor states have a worse heuristic estimate due to the movement costs of the elevator*. In particular, the correct action of moving the elevator up to floor 8 (to deliver the passenger) results in a state of worse heuristic value. If we increased the number of waiting passengers at floor 4, the planning system would therefore try boarding all possible subsets of passengers before moving the elevator. And even once the elevator is moved up to floor 8, the heuristic estimate will only improve after the passenger has been dropped off *and* either (a) the elevator has moved back to floor 4, or (b) the second passenger has boarded and the elevator has moved down to floor 0.

Consequently, movement costs may dominate any progress obtained by transporting passen-

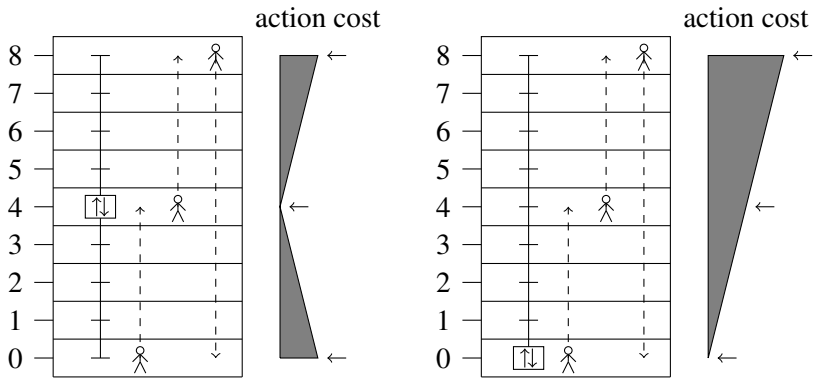


Figure 5.2: Action cost effects in Elevators in a relaxed setting. Travelling 4 floors costs 10, while travelling 8 floors costs 14. Both tasks have the same solution cost (34), but the FF/add heuristic estimates the relaxed cost in the left task as lower (20) than in the right task (24).

gers for a number of successive states. In other words, the planner often has to "blindly" achieve some progress *and* move the elevators towards a middle position given the remaining target floors, in order for the cost-sensitive heuristic to report progress. For the cost-unaware heuristic, the situation is less severe, as the number of elevator movements in the relaxed plan does not increase, and hence the planner encounters a plateau in the search space rather than a local minimum. The use of preferred operators may help to escape from the plateau relatively quickly, whereas a local minimum is much harder to escape from. Two approaches exist that may circumvent this problem. Firstly, the use of enforced hill-climbing rather than greedy best-first search is likely to avoid exploration of the entire local minima: in this approach, a breadth-first search is conducted from the first state of a minimum/plateau until an improving state is found (see Section 2.2.1). Secondly, an improved heuristic could be used that approximates the optimal relaxed cost h^+ more exactly. The cost minimum shown in Figure 5.2 is brought about by the independence assumptions inherent in the FF/add heuristic, which estimate the relaxed cost for each goal fact individually in the cheapest possible way. An *optimal* relaxed plan, however, costs the same in the left and right task. A more accurate approximation of the optimal relaxed cost h^+ could therefore mitigate the described cost minima. Keyder and Geffner (2009) have recently proposed such an improvement of the FF/add heuristic[2] and have shown it to be particularly useful in the Elevators and PARC Printer domains.

Plateaus Due to Capacity Constraints

In general, the relaxed plans in the Elevators domain are often of bad quality. One of the reasons is the way the *capacity* of elevators is encoded in the actions for passengers boarding and leaving elevators. For any passenger p transported in an elevator e, one of the preconditions for p leaving e is that n passengers be boarded in e, where n is a number greater than 0. When constructing a

[2]In Keyder & Geffner's approach, the relaxed plan extracted by the FF/add heuristic is improved by iteratively (1) selecting a fact F, (2) fixing all actions that are not related to F (because they do not contribute to achieving F nor rely on its achievement), and (3) computing a cheaper way of achieving F given the actions that were fixed in the previous step.

relaxed plan, the FF/add heuristic recursively selects actions that achieve each necessary precondition in the cheapest way. This results in boarding *the passenger that is closest to e in the initial state*, even if this passenger p' is different from p, to achieve the condition that some passenger is boarded. The relaxed plan will then contain actions for both boarding p and p' into e, and may furthermore contain other actions for boarding p' into whatever elevator e' is deemed best for transporting p'. Hence, the relaxed plans often contain many unnecessary boarding actions.

As mentioned in Section 5.3, our greedy best-first search breaks ties between equally promising actions by trying the cheaper action first. Consequently, the zero-cost actions for passengers boarding and leaving elevators are tried first in any state. We found that as soon as one passenger is boarded into a certain elevator, the relaxed plans in the next state are often substantially different, in that more passengers are being assigned to that same elevator. This can be explained by the fact that as soon as one passenger is in an elevator, the precondition for leaving that elevator which is having at least one person boarded, is fulfilled (rather than incurring additional cost). In some example tasks we examined, we found that this effect results in committing to bad boarding actions: our planner may initially try some bad boarding action, e. g. boarding the nearest passenger into an elevator to satisfy a capacity precondition for another passenger, as described above. The relaxed plan in the successor state then assigns more passengers to this elevator, at lower cost. Due to the improved heuristic value of the successor state, our planner retains this plan prefix, even though the first action was a bad one. It is plausible (though we did not explore it experimentally) that this effect is stronger for the configurations involving the cost-sensitive heuristic, as the costs of their relaxed plans vary more from one state to the next.

More importantly, the capacity constraints lead to plateaus in the search space, since correct boarding and leaving actions are often not recognised as good actions. For example, if the capacity of an elevator is c, then boarding the first $c - 1$ passengers that need to be transported with this elevator usually leads to improved heuristic values. However, boarding the c-th passenger does not result in a state of better heuristic value if there are any further passengers that need to be transported via the same elevator, because boarding the c-th passenger destroys the precondition that there must be room in the elevator for other passengers to board. Similarly, the correct leaving of a passenger may not lead to an improved heuristic value if it makes the elevator empty and other passengers need to be transported with that elevator later (because the last passenger leaving destroys the precondition for leaving that there must be at least one passenger boarded).

These effects exist for both the cost-sensitive and the cost-unaware FF/add heuristic. However, they typically occur within the plateaus ($\mathbf{F}$) or local minima ($\mathbf{F_c}$) created by the *elevator positions*, as described in the previous section, which means that they affect the cost-sensitive configurations more severely. The plateaus become particularly large when several passengers are waiting on the same floor, e. g. when passengers are accumulating on the floor shared by two blocks in order to switch elevators. The planner then tries to board all possible subsets of people into all available elevators (since the zero-cost boarding and leaving actions are always tried first), moving the elevators and even dropping off passengers at other floors, and may still fail to find a state of

	Solved	**qual. > F**	**qual. < F**
Original tasks	15	3	**10**
Unlimited capacity	29	**17**	6

Table 5.7: Relative qualities of solutions for the cost-sensitive configuration $\mathbf{F_c}$ in the original Elevators domain and in a modified variant of the domain where elevators have unlimited capacity. Shown is the total number of tasks solved by $\mathbf{F_c}$, as well as the number of tasks where it finds a better/worse plan than the cost-unaware configuration **F**.

better heuristic value. When examining the number of states in local minima for each of the configurations, we found that $\mathbf{F_c}$ indeed encounters many more such states than **F**. For example, the percentage of cases in which a state is worse than the best known state is typically around 10% (in rare cases 25%) for **F**. For $\mathbf{F_c}$, on the other hand, the numbers are usually more than 35%, often more than 50%, and in large tasks even up to 80%.

To verify that the capacity constraints indeed contribute to the bad performance of the cost-sensitive heuristic in this domain, we removed these constraints from the IPC tasks and tested the **F** and $\mathbf{F_c}$ configurations on the resulting tasks. Not surprisingly, the tasks become much easier to solve as fewer action preconditions need to be satisfied. More interestingly though, the bad plan qualities produced by $\mathbf{F_c}$ (relative to **F**) indeed become less frequent, as Table 5.7 shows.

In summary, our findings suggest that the bad performance of the cost-sensitive FF/add heuristic in the Elevators domain is due to bad-quality relaxed plans (brought about by the focus on slow elevators and the capacity constraints) and plateaus and local minima in the search space (resulting from the movement costs of elevators and the capacity constraints).

The non-trivial landmarks found in Elevators are: for each passenger p and their target floor f a landmark that specifies that one of the elevators that reach f must stop on that floor, and another landmark that specifies that p must board one of those elevators. These landmarks may not lead to very good search guidance, as they leave open how each passenger gets into the elevator that finally delivers them to their destination. However, while fairly coarse, this heuristic produces plateaus rather than minima in the search space and solves significantly more tasks (20 rather than 15) compared to the FF/add heuristic in the cost-sensitive case. The FF/add heuristic, however, leads to better quality, so that the difference in scores between the two is smaller (11 for $\mathbf{L_c}$ and 9 for $\mathbf{F_c}$). Used together, the strengths of the two heuristics are combined, leading to better results than each of them achieve individually. $\mathbf{FL_c}$ solves 22 tasks and achieves a score of 14.

5.4.2 PARC Printer

The PARC Printer domain (Do et al., 2008) models the operation of a multi-engine printer capable of processing several printing jobs at a time. Each sheet that must be printed needs to pass through several printer components, starting in a feeder and then travelling through transporters, printing engines and possibly inverters before ending up in a finishing tray. The various sheets of a print job must arrive in the correct order at the same finisher tray, but they may travel along different paths

	F	**F$_c$**	**FL**	**FL$_c$**
Tasks solved out of 30	**25**	16	24	23
Avg. solution quality	0.79	**1.00**	0.93	0.95

Table 5.8: Coverage vs. quality in the PARC Printer domain. Average qualities are average IPC scores calculated only on those tasks solved by all shown configurations.

using various printing engines. There are colour printing engines and ones that print in black and white. The action costs are comparatively large, ranging from 2000 to more than 200 000. Colour-printing is the most expensive action, while actions for printing in black and white cost roughly half as much, and actions for transporting sheets are relatively cheap.

Like in the Elevators domain, the cost-sensitive FF/add heuristic did not perform well here, with **F$_c$** failing to solve many of the tasks that the cost-unaware configuration **F** is able to solve. However, in contrast to the Elevators domain, the **F$_c$** configuration finds considerably better plans than **F**. An overview of the number of tasks solved and the average plan quality is shown in Table 5.8. When using landmarks, the differences between cost-sensitive and cost-unaware configurations are strongly reduced, with both landmark configurations achieving a better IPC score than **F**.

Like in Elevators, we found the quality of the relaxed plans to be poor. In the cost-unaware case, a relaxed plan transports sheets from a feeder to the finishing tray via a shortest path, irrespective of whether a suitable printing engine lies on this path. As any path from feeder to finishing tray passes through some printing engine, this frequently involves printing a wrong image on a paper, while additional actions in the relaxed plan handle the transportation from a feeder to a suitable printing engine to print the correct image on the sheet as well. When the cost-sensitive heuristic is used, relaxed plans furthermore become substantially longer, using many transportation actions to reach a cheap printing engine. Analogously to the Elevators domain, the increased complexity associated with longer plans (in combination with the bad quality of the relaxed plans) is thus likely to be the reason for the bad performance of the cost-sensitive heuristic. However, landmarks mitigate the problem, as the numbers of solved tasks in Table 5.8 clearly show. Landmarks found in this domain encompass those for printing a correct image on each sheet, where a disjunctive landmark denotes the possible printers for each sheet. This helps to counteract the tendencies of the cost-sensitive FF/add heuristic to transport sheets to the wrong printers.

In summary, PARC Printer is like Elevators a domain where the cost-sensitive FF/add heuristic performs badly, though in contrast to Elevators the problem here is purely one of coverage, not of solution quality. Even more than in Elevators, landmarks overcome the problems of cost-sensitive $h^{\text{FF/add}}$, so that **FL$_c$** shows similar performance as **F**.

5.4.3 Cyber Security

The Cyber Security domain stands out as a domain where the cost-sensitive configurations **F$_c$** and **FL$_c$** perform significantly *better* than their cost-unaware counterparts. The domain models the

	F	**FL**	**F_c**	**FL_c**
Tasks solved out of 30	**30**	**30**	28	**30**
IPC score	20.44	20.43	24.22	**27.59**

Table 5.9: Coverage and IPC scores in the Cyber Security domain.

vulnerabilities of computer networks to insider attacks (Boddy et al., 2005). The problem consists of gaining access to sensitive information by using various malware programs or physically accessing computers in offices. Action costs model the likelihood of the attack to fail, i. e., the risk of being exposed. For example, many actions in the office of the attacker, like using the computer, do not involve any cost, whereas entering other offices is moderately costly, and directly instructing people to install specific software has a very high associated cost. In particular, action costs are used to model the desire of finding different methods of attack for the same setting. For example, several tasks in the domain differ only in the costs they associate with certain actions.

In the Cyber Security domain, taking action costs into account pays off notably: while the $\mathbf{F_c}$ configuration solves two tasks less than the **F** configuration (see Table 5.9), it nevertheless results in a better total score. Using landmarks, the cost-sensitive configuration is further improved such that it solves all tasks while maintaining the high quality of solutions, resulting in an even larger performance gap between cost-sensitive search and cost-unaware search (see Table 5.9).

The plans found by the cost-unaware search often involve physically accessing computers in other offices or sending viruses by email, and as such result in large cost. Lower costs can be achieved by more complex plans making sophisticated use of software. As opposed to the Elevators and PARC Printer domains, the relaxed plans in Cyber Security are of very good quality. This explains why the performance of the cost-sensitive FF/add heuristic is not negatively impacted by longer plans.

5.4.4 Openstacks

The Openstacks domain models the combinatorial optimisation problem *minimum maximum simultaneous open stacks* (e. g., Fink and Voß, 1999; Gerevini et al., 2009), where the task is to minimise the storage space needed in a manufacturing facility. The manufacturer receives a number of orders, each comprising several products. Only one product can be made at a time, and the manufacturer will always produce the total required quantity of a product (over all orders) before beginning the production of a different product. From the time the first product in an order has been produced to the time when all products in the order have been produced, the order is said to be *open* and requires a *stack* (a temporary storage space). The problem consists of ordering the products such that the maximum number of stacks open at any time is minimised. While it is easy to find *a* solution for this problem (any product order is a solution, requiring n stacks in the worst case where n is the number of orders), finding an *optimal* solution is NP-hard (Linhares and Yanasse, 2002). The minimisation aspect is modelled in the planning tasks via action costs, in that only the action for opening new stacks has a cost of 1, while all other actions have zero cost. This

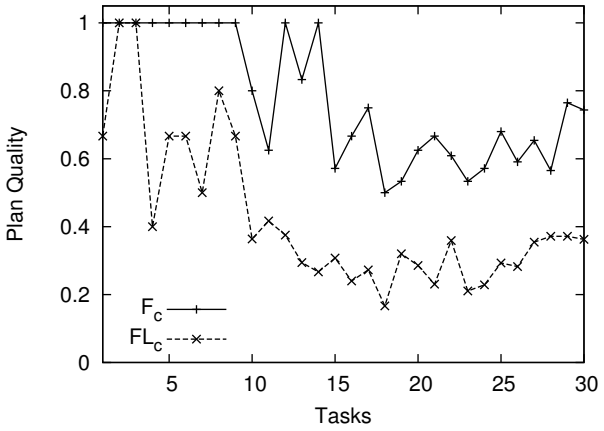


Figure 5.3: Plan quality (measured via the IPC scores) with and without landmarks in the Openstacks domain.

domain was previously used at IPC 2006 (Gerevini et al., 2009). While that earlier formulation of the domain has unit costs, it is equivalent to the cost formulation described above in terms of the relative quality of plans. Since the number of actions that do not open stacks is the same in every plan for a given task, minimising plan length is equivalent to minimising action costs.

We noticed that in this domain using landmarks resulted in plans of substantially lower quality, compared to not using landmarks (see Table 5.1 and Figure 5.3). Across all cost settings, using the landmark heuristic in combination with the FF/add heuristic typically produces plans where the majority of orders is started very early, resulting in a large number of simultaneously open stacks, whereas using *only* the FF/add heuristic leads to plans in which the products corresponding to open orders are manufactured earlier, and the starting of new orders is delayed until earlier orders have been shipped. This is mainly due to the fact that no landmarks are found regarding the opening of stacks, which means that due to the choice of action costs in this domain, all landmarks have cost zero and the landmark heuristic is not able to distinguish between plans of different cost. The landmarks found relate to the starting and shipping of orders as well as the making of products.[3] However, even if landmarks regarding the opening of stacks were found, they would not be helpful: landmarks state that certain things must be achieved, not that certain things need not be achieved. Landmarks can thus not be used to limit the number of open stacks. The landmark orderings are furthermore not helpful for deciding an order between products, as all product orders are possible—which means that no natural orderings exist between the corresponding landmarks—and no product order results in the form of "wasted effort" captured by reasonable landmark orderings.

As mentioned above, all landmarks found have a minimal cost of zero. Therefore, the landmark heuristic fails to estimate the cost to the goal, and distinguishes states only via the *number*

[3]If the size of disjunctions were not limited in our landmark-detection method, it would always find a landmark $\mathit{stacks\_avail}(1) \vee \mathit{stacks\_avail}(2) \vee \cdots \vee \mathit{stacks\_avail}(n)$ stating that at least one of the n stacks must be open at some point. However, any landmark stating that two or more stacks need to be open would require a more complex form of landmarks involving conjunction, which our method does not support.

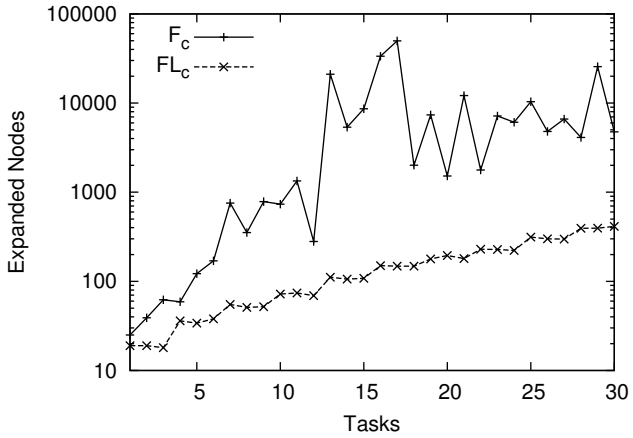


Figure 5.4: Number of expanded search nodes with and without landmarks in the Openstacks domain.

of missing started or shipped orders and products. These goal distance estimates are used directly in **FL** and as tie-breakers amongst the zero-cost landmark-heuristic estimates in $\mathbf{FL_c}$, resulting in the same relative ranking of states by the landmark heuristic in both cases. As soon as one stack is open, for each order o the action that starts o achieves a landmark that is minimal with respect to the landmark orderings (namely the landmark stating that o must be started), and the planner thus tends to start orders as soon as possible. The landmark heuristic is not able to take into account future costs that arise through bad product orderings. This is also a problem for the FF/add heuristic, albeit a less severe one: the FF/add heuristic accounts for the cost of opening (exactly) one new stack whenever at least one more stack is needed, and the heuristic will thus prefer states that do not require any further stacks.

The landmark heuristic does, however, provide a good estimate of the goal distance. Since the landmark heuristic prefers states closer to a goal state with no regard for costs, its use results in plans where stacks are opened as needed. This is reflected in our empirical results, where the additional use of the landmark heuristic drastically reduces the number of expanded search nodes (see Figure 5.4).

Thus, Openstacks is an example of a domain where landmarks are highly detrimental to solution quality. However, using landmarks provides the benefit of speeding up planning by reducing the number of expanded nodes. In this domain, all of our configurations solved all 30 IPC tasks. It is plausible that if there were larger tasks in the benchmark set, the use of landmarks would improve coverage, thus leading to an improvement rather than a degradation in terms of the IPC score.

5.5 Conclusion

We have examined a popular heuristic for cost-based planning, the cost-sensitive FF/add heuristic. In our experimental results, this heuristic achieves some improvement in plan quality, but solves significantly fewer tasks compared to the traditional, cost-unaware FF/add heuristic. When inves-

tigating the reasons for this effect, we found that the cost-sensitive FF/add heuristic reacts strongly to bad relaxed plans, i. e., it is in particular in those domains where the relaxed plans computed by the heuristic have low quality that the cost-sensitive heuristic is likely to perform worse than the cost-unaware heuristic. As we showed for the Elevators domain, action costs may also introduce local minima into the search space where without action costs the search space of the FF/add heuristic would have plateaus. Moreover, the increased complexity of planning for a cheaper goal that is potentially further away from the initial state may lead to worse performance.

Landmarks prove to be very helpful in this context, as they mitigate the problems of the cost-sensitive FF/add heuristic. Using landmarks, the coverage of cost-sensitive search is improved to nearly the same level as that of cost-unaware search, while not notably deteriorating solution quality except in one domain.

Our results suggest that more research into cost-sensitive heuristics is needed. We would like to conduct a more thorough analysis of the short-comings of the cost-sensitive FF/add heuristic, to answer the question whether and how they might be overcome. Keyder and Geffner (2009) propose a method for extracting better relaxed plans from the best supports computed by the cost-sensitive FF/add heuristic, resulting in improved coverage. However, experiments we have conducted with this improved cost-sensitive heuristic show that it still fares worse in terms of the IPC criterion than the cost-unaware FF/add heuristic. It would be interesting to examine to what degree the problems we experienced with the FF/add heuristic extend to other delete-relaxation heuristics, and whether heuristics not based on the delete relaxation could be more effectively adapted to action costs.

6

Preferred Operators and Deferred Evaluation

In the first part of this thesis (Chapters 3–5), we investigated new planning heuristics by developing a range of methods for finding and using landmarks. Another way of improving search approaches to planning is to modify the underlying search algorithms. In the second part of this thesis, we focus on such algorithmic innovations. For example, most successful planning systems based on heuristic search use various search enhancements (see Section 2.2.3). One such enhancement is the use of helpful actions or preferred operators, providing information which may complement heuristic values. A second example is deferred heuristic evaluation, a search variant which can reduce the number of costly node evaluations. Despite the wide-spread use of these search enhancements however, we note that few results have been published examining their usefulness. In particular, while various ways of using, and possibly combining, these techniques are conceivable, no work to date has studied the performance of such variations. In this chapter, we address this gap by examining the use of preferred operators and deferred evaluation in a variety of settings within best-first search. In particular, our findings are consistent with and help explain the good performance of the winner of the satisficing track at the International Planning Competition 2004.

6.1 Introduction

As mentioned in Chapter 2, most winners of the satisficing track in the International Planning Competition since 2000 have been heuristic forward-search planners. There has also clearly been a propagation of ideas in the way that certain aspects of past successful planning systems have been adopted in more recent systems. One example is the use of *helpful actions* in the venerable FF planner, winner of the satisficing track at IPC 2000. Helpful actions are actions that contribute to solving the delete relaxation of a task. As they are likely to also contribute to solving the original task, they can be preferred over actions that are not considered helpful (see Section 2.2.3). This search enhancement, which has been shown to improve FF's performance notably (Hoffmann and Nebel, 2001), has been adapted by the Fast Downward system, winner of the satisficing track at IPC 2004, under the name *preferred operators*. Fast Downward, in turn, inspired many newer planners like Temporal Fast Downward (Röger et al., 2008) and systems used for experiments in various publications (Keyder and Geffner, 2009; Richter et al., 2008; Helmert and Geffner, 2008).

As a result, these systems all use the *deferred evaluation* popularised by Fast Downward. Deferred evaluation is a variant of best-first where the successors of a node are not heuristically evaluated when they are generated, but later when they are expanded (see Section 2.2.3). This may save time if many more nodes are generated than expanded, as heuristic evaluations are computationally costly.

The starting point for this chapter is to note that even though many current planning systems use the search enhancements mentioned above (preferred operators and deferred evaluation), there exists little data on their respective usefulness. For example, the usage of preferred operators (resp. helpful actions) in Fast Downward is different from their usage in FF. While the authors of both planners report significantly improved performance compared to *not* using preferred operators, the question remains which usage is better, and how some of the many other conceivable approaches for exploiting preferred operators would perform. Deferred evaluation, on the other hand, has been suggested to be particularly useful in combination with preferred operators (Helmert, 2006). An open question is whether deferred evaluation is also advantageous in the absence of preferred operators, or when using preferred operators in a different fashion than the one employed by Helmert.

We address these questions by examining the performance of various approaches to using preferred operators, as well as their interaction with deferred evaluation. One finding of this work is that deferred evaluation does not offer any benefit on standard benchmark tasks on average, but leads to strong improvement in some contexts and to strong deterioration in others. On the other hand, we find that the best method for using preferred operators strongly depends on whether deferred evaluation is being employed or not. Using an artificial search space, we demonstrate how the benefit that can be gleaned from preferred operators varies with different aspects of the search space.

6.2 Usages of Preferred Operators

The FF planner uses preferred operators for pruning the search space while Fast Downward uses them in a multi-queue approach (see Section 2.2.3). In both planning systems, preferred operators have been shown to improve performance (Hoffmann and Nebel, 2001; Helmert, 2006). However, the question is open as to which of the two usages leads to better results, assuming that all other aspects of a planner stay fixed. Furthermore, other ways of using preferred operators are conceivable which have not been analysed in the literature to date. An obvious idea is to use preferred operators for tie-breaking and expand, among states of equal heuristic value, preferred successors first. The dual has also been proposed (Vidal, 2004), namely using the heuristic values for tie-breaking among equal preferredness: choose preferred successors whenever they exist, otherwise expand non-preferred successors. Within each of the two groups, choose according to heuristic values. This latter method places more emphasis on preferred operators than the former, but is not as restrictive as expanding only preferred successors like FF does. Finally, the multi-queue

approach may either alternate between preferred-operator queues and regular queues evenly, or may give higher precedence to preferred-operator queues, as described below.

In this chapter, we address the question which of these different usages for preferred operators leads to best performance. The answer may vary for different types of search; we examine greedy best-first search. We look at both the standard implementation of best-first search (in the following called *Eager*), and the deferred-evaluation variant (in the following called *Lazy*). The reason for examining both variants is that they can behave very differently, with Lazy being less informed than Eager, but paying a smaller price for wrong expansions. Hence, the best use of preferred operators may be different for the two. Along the way, we answer the question of whether deferred evaluation is useful in general and whether synergies exist between preferred operators and deferred evaluation.

Boosting preferred operators in Fast Downward. In Section 2.2.3, we described how Fast Downward uses preferred operators with its multi-queue approach by alternating between queues that contain only preferred successors of expanded states and regular queues that contain all successors. The simple alternation method using preferred-operator queues and regular queues equally often has been employed in some work involving Fast Downward (Helmert and Geffner, 2008). The IPC 2004 version of Fast Downward which was used throughout this thesis gives even higher precedence to preferred successors via the following mechanism. The planner keeps a priority counter for each queue, initialised to 0. At each iteration, the next state to be expanded is taken from the queue that has the highest priority. Whenever a state is removed from a queue, the priority of that queue is decreased by 1. If the priorities are not changed outside of this routine, this method alternates between all queues, thus expanding states from preferred queues and regular queues equally often. To strengthen the use of preferred operators, Fast Downward increases the priorities of the preferred-operator queues by a large number *boost* of value 1000 whenever progress is made, i. e., whenever a state is discovered that has a better heuristic estimate than previously expanded states. Subsequently, the next 1000 states will be removed from preferred-operator queues. If another improving state is found within the 1000 states, the boosts accumulate and, accordingly, it takes longer until states from the regular queues are expanded again. While according to Helmert (personal communication) the choice of 1000 as the boost value was ad-hoc, we confirmed that this number is likely to be a good choice. The exact number was not critical in our tests, as we found various values between 100 and 50000 to give similarly good results. Only outside this range did performance drop noticeably. (The experiments with different boost values were conducted using the combination of the FF/add heuristic and landmark heuristic, while all other experiments in this chapter are performed using single heuristics. Given that the exact boost value has little impact here, we assume this would not be different with other heuristics.)

Pseudo-code for Fast Downward's greedy best-first search is shown in Algorithm 6.1 using a set of heuristics H, boosted preferred operators and deferred heuristic evaluation. The main loop (lines 23–34) runs until either a goal has been found (lines 25–27) or the search space has

Global variables:
  $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{O}, C \rangle$ ▷ Planning task to solve
  reg_h, $pref_h$ for $h \in H$ ▷ Regular and preferred open lists
  best_seen_value[] ▷ Best heuristic value seen so far for each heuristic
  priority[] ▷ Numerical priority for each open list

function EXPAND_STATE(s)
  progress ← False
  for $h \in H$ **do**
    $h(s)$, *preferred_ops*(h, s) ← heuristic value of s and pref. operators
    if $h(s) <$ *best_seen_value*$[h]$ **then**
      progress ← True
      best_seen_value$[h] \leftarrow h(s)$
  if *progress* **then** ▷ Boost preferred-operator queues
    for $h \in H$ **do**
      priority$[pref_h] \leftarrow$ *priority*$[pref_h] + 1000$
  for *successor* $\in \{ s[o] \mid o \in \mathcal{O}$ and o applicable in $s \}$ **do**
    for $h \in H$ **do**
      add *successor* to queue reg_h with value $h(s)$ ▷ Deferred evaluation
    if *successor* reached by $o \in$ *preferred_ops*(h, s) by some $h \in H$ **then**
      add *successor* to each pref. op. queue with corresponding $h(s)$

function GREEDY_BFS
  closed_list ← $\emptyset$
  for $h \in H$ **do** ▷ Initialise heuristics
    best_seen_value$[h] \leftarrow \infty$
    for $l \in \{reg, pref\}$ **do** ▷ Regular and preferred lists per heuristic
      $l_h \leftarrow \emptyset$; *priority*$[l_h] \leftarrow 0$
  current_state ← s_0
  loop
    if *current_state* $\notin$ *closed_list* **then**
      if $s = s_\star$ **then**
        extract plan π by tracing current state back to initial state
        return π
      closed_list ← *closed_list* $\cup$ {*current_state*}
      EXPAND_STATE(*current_state*)
    if all queues are empty **then**
      return failure ▷ No plan exists
    $q \leftarrow$ non-empty queue with highest priority
    priority$[q] \leftarrow$ *priority*$[q] - 1$
    current_state ← POP_STATE(q) ▷ Get lowest-valued state from queue q

Algorithm 6.1: Greedy best-first-search with search enhancements.

been exhausted (lines 30–31). The closed list contains all seen states and also keeps track of the links between states and their parents, so that a plan can be efficiently extracted once a goal state has been found (line 26). In each iteration of the loop, the search adds the current state (starting with the initial state) to the closed list and processes it (lines 28–29), unless the state has been processed before, in which case it is ignored (line 24). Then the search selects the next open list to be used (the one with highest priority, line 32), decreases its priority and extracts the next state to be processed (lines 33–34). The processing of a state includes calculating its heuristic values and preferred operators with all heuristics (lines 3–4), expanding it, and inserting the successors into the appropriate open lists (lines 11–15). If it is determined that a new best state has been found (lines 5–7), the preferred-operator queues are boosted by 1000 (lines 8–10).

6.3 Evaluation

We conducted a range of experiments to measure the relative performance of several uses for preferred operators (POs). The usages we compare are

- *No POs*, normal search without preferred operators.
- *PO pruning*, where the search space is restricted to only preferred successor states and the search restarts without POs if it exhausts the restricted search space without a solution. This corresponds to the way of using preferred operators in the FF planner (though FF uses enforced hill-climbing rather than greedy best-first search as the underlying search algorithm).
- *Heur > PO*, where preferred operators are used only to break ties among states of equal heuristic values.
- *PO > Heur*, where preferred operators are used whenever possible, breaking ties with heuristic values. This approach has been used in the YAHSP planner (Vidal, 2004).
- *Dual Queue*, where preferred successors are kept in a second open list in addition to the regular open list, and states are expanded by alternating between the two open lists. This approach has been used in some work involving Fast Downward (Helmert and Geffner, 2008).
- *Boosted Dual Queue*, the dual-queue approach where the open list with preferred successors is boosted by 1000 whenever progress is made during the search, leading to more preferred successors being expanded. This approach was used in the IPC 2004 version of Fast Downward.

We conducted experiments on all planning tasks from past International Planning Competitions between 1998 and 2006 except for the trivial Movie domain, totalling 1582 tasks. Four different planning heuristics were used: the FF/add heuristic, the context-enhanced additive (cea)

heuristic, the causal graph (CG) heuristic, and the additive heuristic. For details on these heuristics, see Section 2.2.2. We follow the Fast Downward implementation of the FF/add heuristic by defining preferred operators to be those applicable actions which appear in the relaxed plan. By contrast, FF's "helpful actions" additionally include all applicable actions which add a *precondition* of an action in the relaxed plan that is not true in the current state. For the additive heuristic we generate preferred operators in the same way as for the FF/add heuristic. (Without referring to relaxed plans, these preferred operators can be characterised as those applicable actions which contribute to the heuristic value, i.e. we select a best supporting action for the goal, according to the heuristic, and, recursively, a best supporting action for unsatisfied preconditions of selected actions.) The preferred operators in the CG and cea heuristics are those applicable actions that change the current value of a goal variable to a value that is on a lowest-cost path (as computed by the heuristic) to the goal value in the domain transition graph of that variable. If no such action exists for a goal variable, the process recurses for variables that are involved in conditions for changing the value of the goal variable (Helmert, 2006).

The experiments were conducted on a heterogeneous cluster of Intel Xeon and AMD Opteron CPUs, ranging from 2.2 GHz to 2.83 GHz. (The magnitude of the experiments precluded experiments on homogeneous machines.) To allow fair comparisons, for each given planning task all planner configurations using the same heuristic were run on the same CPU. The timeout was 30 minutes and the memory limit 1.75 GB in all cases.

For each planner configuration, we report results regarding coverage, the number of heuristic state evaluations, the runtime and the quality of the plans found. We report evaluations rather than expansions to be able to compare the two search types Eager and Lazy in a meaningful way. While the cost for an evaluation is the same for both search variants, the cost of an expansion differs greatly for the two. To expand a state, Lazy evaluates only the state itself and adds pointers to the open list, while Eager generates and evaluates all successor states. For Lazy, evaluations and expansions are the same; for both search types the number of evaluations is the number of states actually generated. Most importantly, roughly 80% of the total runtime of the planner is spent calculating heuristic values, making evaluations a platform-independent indicator of the search effort.

In order to facilitate the comparison of algorithms regarding various performance criteria, we measure performance for all criteria via scores from the range 0–100, where best possible performance in a task counts as 100, while failure to solve a task and worst performance are counted as 0. For coverage, a solved task counts as 100. Plan quality is measured using the performance criterion of IPC 2008 (see Section 2.4) scaled by the factor 100.[1]

Evaluations and runtime are measured on a log scale due to the exponential nature of the problem. Performance better than a lower limit (100 states for evaluations and 1 second for runtime)

[1]Note that this criterion in fact contains information about both plan quality and coverage. However, the alternative would be to rate plan quality only on those tasks solved by all compared configurations. This would exclude many tasks, since due to the large number of configurations we do not want to compare configurations pair-wise here.

counts as 100, performance worse than an upper limit (1,000,000 states for evaluations and the timeout of 30 minutes for runtime) counts as 0. In between, we interpolate with a logarithmic function. The scores reported for domains are averaged over the tasks in the domain; when summarising results from multiple domains we show normalised averages such that each domain is weighed equally. (Thus, a coverage score of e. g. 74.03 means that the configuration solves on average 74.03% of the tasks in a randomly picked domain.)

Table 6.1 shows the results for all configurations and heuristics.[2] We begin by discussing the coverage. The first thing we note is that the use of preferred operators almost always improves performance, and that the impact of preferred operators is significantly larger even than the choice of heuristic. With suitable usage of preferred operators, the weakest heuristics (CG and additive) perform better than the strongest heuristics without preferred operators (FF/add and cea). Which usage of preferred operators gives best results is dependent on the type of search: for eager search the simple dual-queue approach works best, for deferred evaluation the boosted dual-queue is best. The results are remarkably consistent across the four heuristics. We now discuss the various options in turn.

Using preferredness as a tie-breaker among states of equal heuristic value ("Heur > PO") is a relatively subtle way of using preferred operators. Compared to not using preferred operators, this option usually improves performance slightly for Eager and notably for Lazy. However, for both search types the results are far from those obtainable via other options. Using preferred operators whenever possible ("PO > Heur") leads with one exception to more improvement than "Heur > PO" for both search types. For Lazy, this option works very well, giving consistently second- or third-best results among the 6 possible PO options. For Eager, this configuration does not work quite as well and usually ranks fourth. PO pruning is similarly good as "PO > Heur" for Lazy, and better than that option for Eager, ranking second-best on average. In particular, when using PO pruning the performance of the two search types is comparable. The simple dual-queue approach is the best option for Eager, but gives mixed results for Lazy, ranging from second-best (CG heuristic) to fourth-best (cea and additive heuristic). The boosted dual-queue approach is the best option for Lazy, but for Eager the performance is notably worse than that of the simple dual queue.

Summarising the results, it seems that for Eager a certain amount of guidance via preferred operators is useful (e. g., "Heur > PO" or dual-queue), but strong use of preferred operators can be counter-productive (e. g., "PO > Heur" and the boosted dual-queue method are worse than the simple dual-queue method). When using deferred evaluation, however, stronger use of preferred operators always seems to improve results. PO pruning works well for both search types, probably due to the significantly reduced branching factor, as only preferred successors are evaluated (unless the restricted search fails, which we found to be rare). Note that for Lazy, PO pruning and "PO >

[2]The slight differences compared to the results in Chapter 4 are due to running these experiments on different machines and to small evolutions in Fast Downwards code. The difference to the results published previously (Richter and Helmert, 2009) is due to the fact that we excluded the trivial Movie domain here.

Search	PO Use	Coverage Score	Eval. Score	Quality Score	Time Score
FF/add Heuristic					
Lazy	No POs	73.19	52.86	67.40	66.59
	Heur > PO	78.43	61.72	71.88	71.39
	PO > Heur	83.18	69.81	74.72	78.41
	PO Pruning	83.51	69.57	75.36	78.83
	Dual Queue	82.90	63.98	76.50	75.37
	B. Dual Queue	**86.31**	**71.70**	**78.27**	**81.36**
Eager	No POs	74.27	50.88	71.22	67.05
	Heur > PO	75.02	52.80	71.63	68.32
	PO > Heur	80.82	56.85	76.21	73.84
	PO Pruning	84.14	**67.09**	78.89	**79.09**
	Dual Queue	**86.46**	59.65	**82.87**	78.50
	B. Dual Queue	83.13	57.26	78.64	75.18
Cea Heuristic					
Lazy	No POs	73.22	53.51	67.04	64.98
	Heur > PO	74.56	59.36	68.38	67.42
	PO > Heur	83.04	67.37	74.02	75.74
	PO Pruning	82.65	67.01	74.48	75.61
	Dual Queue	79.50	62.51	72.83	71.80
	B. Dual Queue	**85.80**	**69.27**	**76.87**	**78.13**
Eager	No POs	73.18	50.44	70.01	63.90
	Heur > PO	73.16	50.90	70.11	64.14
	PO > Heur	81.32	55.08	75.76	70.84
	PO Pruning	82.95	**65.08**	77.46	**75.68**
	Dual Queue	**84.75**	57.72	**80.97**	74.05
	B. Dual Queue	83.59	55.50	77.72	71.69
CG Heuristic					
Lazy	No POs	70.22	48.40	64.07	63.36
	Heur > PO	72.09	52.81	64.47	65.45
	PO > Heur	72.52	53.23	65.86	65.54
	PO Pruning	71.49	50.82	65.07	64.26
	Dual Queue	74.30	**55.13**	67.15	67.67
	B. Dual Queue	**75.62**	55.06	**68.71**	**68.43**
Eager	No POs	70.91	46.22	67.54	63.04
	Heur > PO	71.07	46.49	67.67	63.32
	PO > Heur	70.29	45.65	66.76	63.15
	PO Pruning	71.50	**48.60**	66.84	63.56
	Dual Queue	**74.96**	48.04	**71.90**	**66.48**
	B. Dual Queue	70.89	45.68	67.35	63.36
Additive Heuristic					
Lazy	No POs	69.90	49.05	62.78	62.77
	Heur > PO	72.95	55.90	65.50	66.03
	PO > Heur	83.03	65.90	74.40	76.85
	PO Pruning	83.55	65.81	74.99	77.47
	Dual Queue	77.47	58.45	69.86	69.81
	B. Dual Queue	**85.09**	**67.95**	**76.71**	**79.28**
Eager	No POs	70.67	47.38	66.83	63.09
	Heur > PO	71.36	48.33	67.19	63.84
	PO > Heur	82.82	54.46	77.40	73.21
	PO Pruning	**84.39**	**64.12**	79.04	**77.91**
	Dual Queue	83.73	55.68	**79.45**	74.68
	B. Dual Queue	84.21	54.92	78.88	74.28

Table 6.1: Summary of results. Scores are in the interval 0–100; higher scores are better. The best PO use for each search type and heuristic is indicated in bold. Results are reported as normalised averages that weigh all domains equally. Details on the scoring mechanisms can be found in the text.

H" give similar results as they lead to identical behaviour in those cases where a goal can be found by only expanding preferred successors. For Eager, however, "PO > H" is worse than PO pruning because with "PO > H" Eager evaluates all successors, whereas with PO pruning it evaluates only preferred successors.

Comparing Eager with Lazy, we note that Eager performs better in most of the configurations; with boosted dual queue, however, deferred evaluation is better. The best performance obtainable from each search variant (using the dual-queue setting for Eager, and boosted dual-queue for Lazy) is very similar. Lazy consistently evaluates fewer states than Eager and is usually faster. Eager, on the other hand, finds plans of better quality. This is not surprising as Eager is better informed with respect to heuristic values than Lazy. For both search variants, plan quality is improved when using preferred operators.

6.3.1 Interactions Between Search Type and PO Use

When using deferred evaluation, all successors of a state are placed in the open list with the same heuristic value (that of the parent). Preferred operators are thus particularly useful for lazy search as they can help recognise the best successor of a state. This is why stronger use of preferred operators consistently improves results for Lazy. Eager search, on the other hand, evaluates all successors of a state at once. Eager search thus has an estimate (through the heuristic values of the successors) of which successors are best, and preferred operators may not provide as much additional information to Eager than to Lazy. Preferred operators can be helpful for choosing among states of equal heuristic value, for example, as they act as a second type of information about which states likely lie on a path to the goal. However, preferred operators may even be detrimental to the performance of Eager if they are ill-informed (i. e. if successors are deemed preferred even though they are in fact worse than some of their siblings).

6.3.2 Summary of Findings

We summarise the main findings that hold on average, as shown by Table 6.1.

- **Preferred operators are very helpful.** For both search types, an average coverage improvement of 10-15% can be obtained with the respective best PO usage; this is far more than the differences between the various heuristics.

- **Best PO use depends on the search type.** While both eager and lazy search are improved by a similar amount through using POs, Eager performs best with PO pruning and the standard dual-queue approach, while Lazy excels when using POs more strongly via the boosted dual queue.

- **Deferred evaluation trades time for quality.** Both Eager and Lazy obtain similar coverage. But Lazy is faster while Eager finds better plans. Our work thus confirms a claim by Helmert (2006) which had been lacking empirical support to date.

	Lazy Search						Eager Search					
	No PO	H > PO	PO > H	Prun.	DQ	B. DQ	No PO	H > PO	PO > H	Prun.	DQ	B. DQ
Assembly: heur = FF/add												
Coverage	63.33	66.67	86.67	86.67	96.67	100.00	56.67	56.67	80.00	86.67	100.00	90.00
Eval.	40.02	42.88	82.00	82.00	81.27	87.75	29.29	29.94	55.31	74.85	70.52	55.80
Quality	61.63	64.97	84.99	84.99	93.55	97.19	55.13	55.44	78.61	85.14	97.25	87.14
Time	59.57	59.33	86.61	86.61	93.34	98.73	51.05	51.27	78.84	85.19	100.00	84.07
Openstacks: heur = cea												
Coverage	83.33	86.67	90.00	93.33	90.00	86.67	76.67	76.67	90.00	90.00	86.67	90.00
Eval.	43.50	43.63	62.07	62.44	50.22	61.37	35.25	35.23	59.76	61.49	50.35	59.76
Quality	82.79	85.83	89.59	92.93	89.16	86.26	76.12	76.12	89.68	89.68	86.34	89.68
Time	66.72	67.20	79.98	79.83	72.34	78.86	58.70	58.63	79.17	79.79	72.97	78.89
TPP: heur = additive												
Coverage	63.33	76.67	96.67	96.67	80.00	100.00	63.33	70.00	93.33	93.33	100.00	90.00
Eval.	33.50	40.19	67.17	65.78	44.57	68.44	33.14	35.09	58.33	64.15	62.25	58.12
Quality	58.33	70.36	92.89	92.55	73.34	96.04	56.33	62.51	89.54	87.91	91.51	86.30
Time	50.48	58.98	81.99	80.01	62.08	82.99	50.17	52.57	76.85	80.53	82.04	76.91
Pipesworld Tankage: heur = additive												
Coverage	36.00	38.00	76.00	74.00	44.00	72.00	34.00	34.00	80.00	80.00	54.00	78.00
Eval.	20.40	25.46	54.21	55.89	27.48	52.61	19.98	19.46	42.61	58.09	30.05	42.28
Quality	30.06	33.43	57.13	55.35	38.17	56.58	31.62	31.59	64.24	63.93	50.41	62.81
Time	25.26	29.62	57.08	60.66	31.22	54.74	25.45	23.99	51.89	62.92	36.20	50.79
Storage: heur = CG												
Coverage	53.33	60.00	70.00	56.67	63.33	70.00	53.33	60.00	60.00	53.33	60.00	60.00
Eval.	44.28	46.66	58.53	43.97	52.54	58.09	43.26	44.73	47.60	43.11	48.94	47.60
Quality	50.93	53.30	66.61	53.70	61.75	66.28	50.91	56.02	56.31	50.68	56.96	56.31
Time	52.67	54.94	65.84	53.03	58.91	65.40	52.61	54.84	59.03	52.39	59.31	59.05
PSR Large: heur = additive												
Coverage	58.00	66.00	46.00	50.00	64.00	64.00	58.00	60.00	46.00	50.00	54.00	48.00
Eval.	36.95	53.03	29.09	31.88	50.67	42.64	37.46	37.89	24.81	31.93	31.45	25.41
Quality	50.88	61.15	39.75	44.61	57.25	56.22	51.41	52.44	41.72	45.51	49.46	42.64
Time	54.18	61.63	42.32	48.72	60.56	58.75	54.69	55.50	41.81	48.77	50.08	43.09
Philosophers: heur = FF/add												
Coverage	100.00	100.00	43.75	43.75	100.00	81.25	100.00	52.08	35.42	47.92	100.00	37.50
Eval.	37.87	38.31	19.80	19.80	38.37	26.62	37.61	23.73	15.73	19.92	37.67	15.98
Quality	100.00	100.00	29.24	29.24	97.87	66.04	100.00	52.08	23.49	30.91	100.00	25.81
Time	74.46	73.96	36.07	36.05	73.79	58.16	75.58	44.52	31.06	38.56	73.61	32.32
Schedule: heur = FF/add												
Coverage	18.00	16.00	86.67	86.67	75.33	99.33	22.00	16.67	75.33	88.00	100.00	76.00
Eval.	12.65	15.19	72.85	72.36	46.39	79.18	14.27	12.00	33.42	67.35	46.25	33.42
Quality	17.09	15.53	79.12	79.31	67.12	90.05	21.55	16.32	70.77	81.46	94.65	71.43
Time	17.51	15.79	82.81	82.38	62.19	93.95	21.25	16.26	61.96	82.69	84.50	62.17
Satellite: heur = FF/add												
Coverage	69.44	100.00	100.00	100.00	88.89	100.00	69.44	72.22	77.78	91.67	77.78	77.78
Eval.	38.59	79.21	81.34	81.34	66.98	81.24	36.74	40.53	43.11	70.78	42.05	43.11
Quality	63.70	94.47	94.33	94.33	83.76	94.33	68.48	71.49	76.18	89.85	76.77	76.18
Time	57.24	81.02	82.29	82.53	74.12	82.99	57.97	60.77	65.59	78.30	64.82	65.63
Logistics 1998: heur = CG												
Coverage	100.00	94.29	91.43	94.29	100.00	97.14	97.14	97.14	85.71	94.29	97.14	88.57
Eval.	68.66	66.96	63.12	65.08	71.51	64.37	49.51	49.30	40.02	48.69	47.87	40.23
Quality	98.20	87.53	84.33	92.61	92.97	89.66	96.37	95.54	80.21	93.91	95.38	82.94
Time	89.39	85.14	81.66	84.60	90.17	83.06	75.34	75.53	65.52	74.17	74.03	65.50
Pathways: heur = cea												
Coverage	26.67	30.00	63.33	63.33	40.00	63.33	33.33	30.00	93.33	90.00	93.33	93.33
Eval.	18.48	21.51	49.68	49.68	26.56	49.31	17.90	19.27	47.43	62.65	45.92	47.43
Quality	25.67	29.28	60.93	60.93	38.26	60.89	33.14	29.49	92.22	88.70	92.24	92.22
Time	25.08	28.02	62.91	62.92	36.42	62.50	27.28	27.16	84.05	87.61	83.04	84.06

Table 6.2: Detailed results for selected domains.

6.3.3 Details and Special Cases

In Table 6.2, we report individual results for a number of domain/heuristic combinations. The top part of the table contains "typical" cases that reflect the general findings discussed above. Preferred operators increase performance notably, and lazy and eager search perform similarly well. While there is some variation (e. g. in Pipesworld Tankage), in general it holds that "H > PO" is slightly better than not using POs, while the other PO usages are noticeably better; and PO pruning or dual-queue usage is best for Eager while the boosted dual-queue approach is best for Lazy.

The rest of the table contains exceptions from the general trend, which we discuss in turn. Firstly, we examine cases which are outliers with regard to PO use. In PSR Large with the additive or FF/add heuristic, we find that both search types gain some improvement from subtle use of preferred operators in the "H > PO" setting. However, stronger use of POs is detrimental, which becomes particularly evident in the settings "PO > H" and PO pruning. The reason seems to be that few preferred operators are found in this domain; this holds for both the additive and the FF/add heuristic. The setting "H > PO" uses the information about preferred operators only when it exists and adds to the knowledge about heuristic values, and thus improves results. By contrast, restricting the search to POs means that we may often be precluded from expanding the current best state because it may not happen to be preferred.

The Philosophers domain with the additive or FF/add heuristic is interesting because even the subtle use of POs via "H > PO" is detrimental for eager search, and with the exception of the dual-queue approach, all strong uses of POs lead to worse performance for both search types. When examining the domain, we found that it contains large plateaus or near-plateaus, and many ill-informed preferred operators. In large areas of the search space 80–100% of the operators are preferred, of which only 5–10% lead to states with better heuristic values. Breadth-first search may be the best way to escape from plateaus, and when not using POs, this is what our search does (due to the open list treating states of equal heuristic value on a first-in-first-out basis). When using the dual queue, the search performs breadth-first search half of the time, which still works very well. By contrast, strong use of POs means that the search spends most of the time following ill-informed POs, and by *not* doing breadth-first search it may not manage to leave the plateaus as quickly.

In Schedule with the FF/add heuristic, the use of POs in a dual-queue approach is a substantial improvement over not using POs; however, the subtle use of POs as tie-breakers in "H > PO" is notably worse than not using POs. While we are at present not sure what causes this behaviour, one possible explanation is that POs are helpful in some parts of the search space but useless or even detrimental in others.

It is noteworthy that the dual-queue approach may be the most "robust" way of using preferred operators. For all other PO uses, bad cases exist where they perform significantly worse than not using POs. By contrast, the dual-queue approach always performs fairly well and seems to be able to make use of helpful POs while not getting overly distracted by ill-informed POs.

Lastly, the third part of Table 6.2 contains examples where the performances of the two search

	Lazy			Eager		
inst.	No PO	Prun.	DQ	No PO	Prun.	DQ
25	150K	1K	2K	71K	2K	71K
26	211K	1K	3K	116K	3K	80K
27	24K	1K	2K	96K	3K	96K
28	46K	2K	3K	195K	6K	195K
29	—	1K	3K	—	7K	285K
30	—	3K	6K	—	10K	420K
31	—	4K	5K	—	14K	—
32	—	10K	—	—	28K	—
33	—	8K	—	—	41K	—
34	—	4K	7K	265K	15K	265K
35	—	10K	—	—	26K	—
36	—	7K	—	—	27K	—

Table 6.3: Evaluations in Satellite with the cea heuristic.

variants Eager and Lazy differ substantially. In Satellite and Logistics, a large branching factor in combination with an informative heuristic leads to lazy search being extremely useful, so that it dominates eager search notably with respect to coverage, evaluations and runtime. We will discuss this effect in Satellite in more detail below. On the other hand, Pathways is a domain where deferred evaluation seems to be particularly harmful, and Eager performs significantly better than Lazy.

Table 6.3 provides a closer look at the effect of deferred evaluation in the Satellite domain when using the cea heuristic (results for the other heuristics are similar). We show the number of evaluations for the last 12 instances of the domain. With the configurations that are not shown (H > PO, PO > H, and boosted dual queue), Lazy performs exactly the same as with PO pruning, while Eager performs no better than without POs. Eager obtains best performance with PO pruning in this domain as the pruning greatly reduces the large branching factor of the search space. However, even in this setting Lazy outperforms Eager notably with regards to evaluations and, consequently, runtime. Table 6.3 shows that with exception of the first two instances, Lazy evaluates substantially fewer states than Eager, with differences as high as two orders of magnitude in some cases. In instance 34, eager search with the dual queue finds a plan of length 287 expanding only 287 states, i. e., it is guided straight to the goal. However, due to the large branching factor it still needs to evaluate more than 265,000 states, and runs for 29 minutes, while Lazy only evaluates 4,000 states and terminates after 43 seconds.

6.4 A Controlled Experiment

To more closely analyse how the benefit from preferred operators varies with different characteristics of the search space, we conducted a set of experiments on a manually-designed search space. We chose the parameters of this artificial search space without much experimentation, as it was straightforward to find settings that show informative behaviour, i. e. where the task is not too easy and not too hard. Other settings are possible and may often lead to similar results. Our goal was

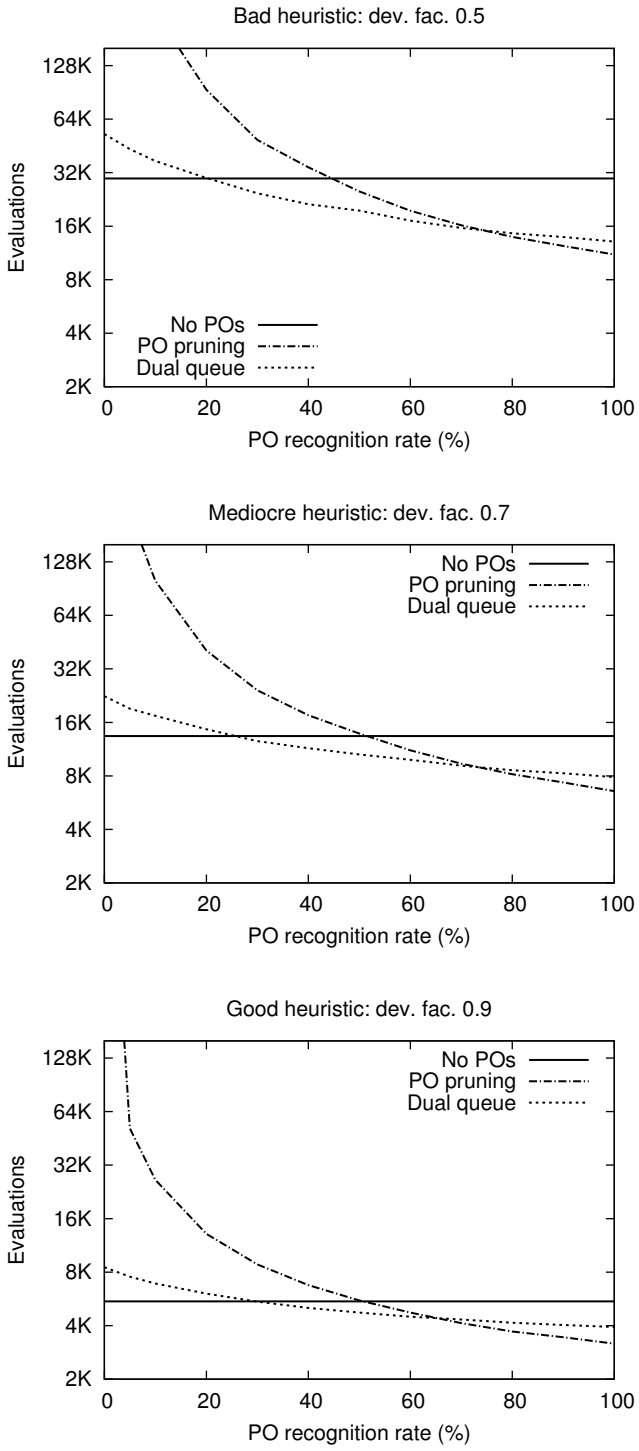


Figure 6.1: Experiments in an artificial search space. On the x-axis we vary the rate at which good actions are recognised (labelled as preferred operators). Results are shown for three different levels of heuristic quality: rather uninformative, moderately informative and very informative.

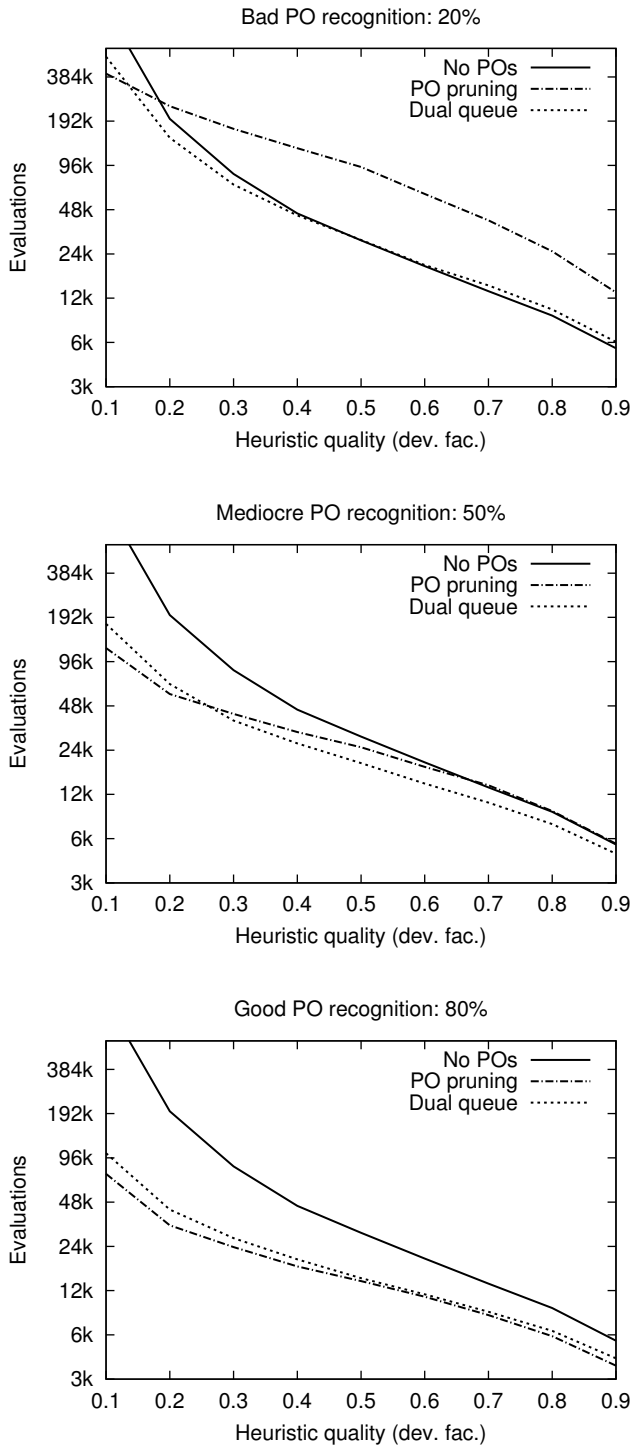


Figure 6.2: Experiments in an artificial search space. On the x-axis we vary the quality of the heuristic. Results are shown for three different rates of PO recognition: recognising 20%, 50% and 80% of the useful actions in a state as preferred operators.

to create a scenario where preferred operators can be useful but also harmful, depending on how well the preferredness of an operator predicts that it actually leads to a better state, and how well informed the heuristic is in comparison. The parameters we vary are:

- The quality of the heuristic. We use an admissible heuristic that randomly deviates by up to a factor *dev. fac.* ≤ 1 from the approximate goal distance *agd* (see below) of a state. We experimented with values from 0.1 to 0.9. in 0.1-step increases.

- The recognition rate *rec. rate* for preferred operators, i. e., in what percentage of cases a useful action (one that leads to a better state) is labelled as a preferred operator. For this parameter, we tested values between 0% and 100% using steps of 10%.

We fixed a branching factor of 25 and a typical solution depth of 50. States are characterised by their *approximate goal distance (agd)*. The initial state has an *agd*-value of 50 and states with a value of 0 are labelled as goals. For each state with *agd*-value d, we generate the values of the 25 successors randomly as follows: with probability 0.04 they are $d - 1$, with probability 0.16 they are $d + 1$, and with probability 0.8 they are d. This means that there exist on average one successor with a lower *agd*-value, 20 successors with equal *agd*-values, and 4 successors with higher *agd*-values than the parent. Likewise, the decision whether or not an action is labelled as a preferred operator is made randomly: if an action is useful, i. e. if it leads to a successor state with better *agd*-value, it is preferred in the percentage of cases given by *rec. rate*; else it is preferred in 10% of the cases. If the recognition rate decreases, this means that the relative chance of a preferred operator *not* being useful increases.

Figures 6.1 and 6.2 show the number of evaluated states for the two search algorithms in our experiment. In Figure 6.1, we fixed the heuristic quality at three different values (low in the top panel, medium in the centre and high on the bottom), and varied the recognition rate of preferred operators. The numbers shown are averaged over 100 runs with different random seeds. As can be seen, a low recognition rate leads to preferred operators being detrimental to the search, while with a high recognition rate, using preferred operators improves performance compared to not using them. It is also evident that PO pruning suffers more strongly from ill-informed preferred operators than the dual-queue approach. The relative performance of the approaches shown in Figure 6.1 is notably independent of the quality of the heuristic. The dual-queue approach typically becomes better than not using POs if the recognition rate is more than 20–30%; PO pruning has a higher threshold of 40–50%.

Figure 6.2 provides a different look on the same data, where we fix the recognition rate of preferred operators and vary the heuristic quality. As can be seen in the centre and bottom panels, the advantage of using preferred operators decreases for high values of heuristic quality, i. e., preferred operators are most useful if the heuristic is not well informed. But even given very high heuristic quality, POs can still notably improve performance if the recognition rate is high.

6.5 Conclusion

We have examined two search enhancements for planning that are widely used, preferred operators and deferred evaluation. Our findings include that the obvious method of using preferred operators as tie-breakers provides little benefit, compared to other approaches; and that the dual-queue approach, which keeps preferred operators in an additional open list, performs best amongst all tested approaches in a standard best-first search. In particular, the dual-queue approach dominates pruning, the method that was proposed when preferred operators were first introduced. With controlled experiments in an artificial search space, we showed that pruning performs badly if the preferred operators are ill-informed. Our results thus suggest that FF might be improved by using the dual-queue approach, though experiments with FF's enforced hill-climbing search algorithm and FF's definition of preferred operators would be needed to confirm this hypothesis.

Our work confirms previous claims that deferred evaluation can be very useful in the presence of large branching factors and that this method trades plan quality for time. Our findings also help to explain the good performance of Fast Downward at IPC 2004. This planner used deferred evaluation in combination with the boosted dual-queue option, a combination which led to best performance in our experiments. However, the conceptually simpler approach of using standard best-first search with the unboosted dual queue performs equally well on average, albeit having different strengths and weaknesses. Our results show that boosting the dual queue in Fast Downward should always be done for best results, but it would be detrimental in a standard best-first search.

7

Restarts for Anytime Planning

In this chapter, we examine *anytime planning* as a way of finding good solutions given limited time and uncertainty about the hardness of the task at hand. Anytime search algorithms solve optimisation problems by quickly finding a (usually suboptimal) first solution and then finding improved solutions when given additional time (see Section 1.1). To deliver an initial solution quickly, they are typically greedy with respect to the heuristic cost-to-go estimate h. In this chapter, we show that this low-h bias can cause poor performance if the greedy search makes early mistakes. Building on this observation, we present a new anytime approach that restarts the search from the initial state every time a new solution is found. We demonstrate the utility of our method via experiments in planning as well as other domains, and show that it is particularly useful for problems where the heuristic has systematic errors.

7.1 Introduction

As described in Section 2.2.1, the A* algorithm can be used to find optimal solutions for shortest-path problems, given an admissible heuristic and sufficient resources. However, in large tasks we may not be able to afford the resources necessary for calculating an optimal solution. A popular approach in this circumstance is complete anytime search, in which a (typically suboptimal) solution is found quickly, followed over time by a sequence of progressively better solutions, until the search is terminated or solution optimality has been proven.

Weighted A*, or short WA*, uses a weighting factor $w > 1$ and searches more greedily the larger w is, at each step selecting a state for expansion that minimises the function $f'(s) = g(s) + w \cdot h(s)$ (see Section 2.2.1). The ability to balance speed against quality while providing bounds on the suboptimality of solutions makes WA* an attractive basis for anytime algorithms (Hansen et al., 1997; Likhachev et al., 2004; Hansen and Zhou, 2007; Likhachev et al., 2008). For example, WA* can be run first with a very high weight, essentially resulting in a greedy best-first search that tries to find a solution as quickly as possible. Then the search can be continued past the first solution to find better ones. By reducing the weight over time, the search can be made progressively less greedy and more focused on quality.

Complete anytime algorithms that are not based on WA* have also been proposed (Zhang,

1998; Zhou and Hansen, 2005; Aine et al., 2007). In these cases, an underlying global search algorithm is usually made more greedy and local by initially restricting the set of states that can be expanded, thus giving a depth-first tendency to the search which can lead to finding a first solution quickly. Whenever a solution is found, the restrictions are relaxed, allowing for higher-quality solutions to be found.

We start by noting that the greediness employed by anytime algorithms to quickly find a first solution can also create problems for them. If the heuristic goal distance estimates are inaccurate, for example, the search can be led into an area of the search space that contains no goal or only poor goals. Anytime algorithms typically *continue* their search after finding the first solution, rather than starting a new search. This seems reasonable as it avoids duplicate effort. However, a key observation that we will discuss in detail below is that continued search may perform badly in problems where the initial solution is far from optimal and finding a significantly better solution requires searching a different part of the search space. In such cases *restarting* the search can be beneficial, as it allows changing bad decisions that were made near the initial state more quickly. Reconsidering these early decisions means the quality of the best-known solution may improve faster with restarts than if we continued to explore the area of the search space around the previous solution.

Building on this observation that forgetfulness can be bliss, we propose an anytime algorithm that incorporates helpful restarts while retaining most of the knowledge discovered in previous search phases. Our approach differs substantially from existing anytime approaches, as previous work on anytime algorithms has concentrated explicitly on *avoiding* duplicate work (Likhachev et al., 2004, 2008). We show that the opposite can be useful, namely discarding information that may be biased due to greediness. We show that our method outperforms existing anytime algorithms in planning and that it is competitive in other benchmark domains. Using an artificial search space, we elucidate conditions under which our approach performs well.

7.2 Previous Approaches

We first present some existing approaches and discuss the circumstances under which their effectiveness may suffer.

Anytime Algorithms Based on Weighted A*

Anytime algorithms based on WA* run WA* but do not stop upon finding a solution. Rather, they report the solution and continue the search, possibly adjusting some of their search parameters. A straightforward adjustment, for example, is to use the cost of the best known solution for pruning: after we have found a solution of cost c, we do not need to consider any states that are guaranteed to lead to solutions of cost c or more. This is the approach taken by Hansen and Zhou in their Anytime WA* algorithm (Hansen and Zhou, 2007). Updating the cost bound is the only adjustment

in Anytime WA*; the weight w is never decreased. (While the authors discuss this option, they did not find it to improve results on their benchmarks.)

The Anytime Repairing A* (ARA*) algorithm by Likhachev et al. (2004, 2008) assumes an admissible heuristic and reduces the weight w each time it proves w-admissibility of the incumbent solution. This implicitly prunes the search space as no state is ever expanded whose f'-value is larger than that of the incumbent solution. When reducing w, ARA* updates the f'-values of all states in the open list according to the new weight. Furthermore, ARA* avoids re-expanding states within search phases (where we use "phase" to denote the part of search between two weight changes). Whenever a cheaper path to a state is discovered, and that state has already been expanded in the current search phase, the state is not re-expanded immediately. Instead it is suspended in a separate list, which is inserted into the open list only at the beginning of the next phase. The rationale behind this is that even without re-expanding states the next solution found is guaranteed to be within the current sub-optimality bound (Likhachev et al., 2004, 2008). Of course, it may be that this solution is worse than it would have been had we re-expanded.

Other Anytime Algorithms

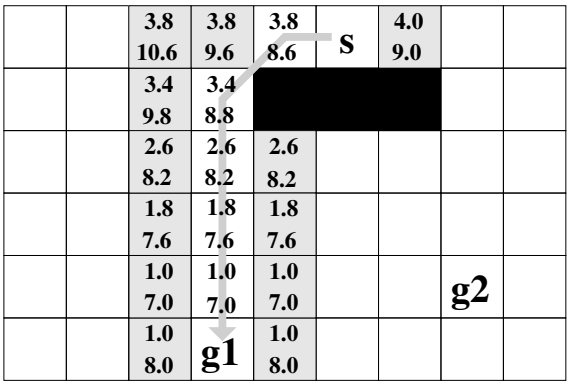
Recent anytime algorithms that are not based on WA* include the A*-based Anytime Window A* algorithm (Aine et al., 2007) and beam-stack search (Zhou and Hansen, 2005) which is based on breadth-first search. In both cases, the underlying global search algorithm is made more greedy by initially restricting the set of states that can be expanded. In Window A*, this is done by using a "window" that slides downwards in a depth-first fashion: whenever a state with a larger g-value (level) than previously expanded is expanded, the window slides down to that level of the search space. From then on, only states in that level and the k levels above can be expanded, where k is the height of the window. Initially k is zero and it is increased by 1 every time a new solution is found. Anytime Window A* can suffer from its strong depth-first focus if the heuristic estimates are inaccurate.

In beam-stack search, only the b most promising nodes in each level of the search space are expanded, where the beam width b is a user-supplied parameter. The algorithm remembers which nodes have not yet been expanded and returns to them later. However, beam-stack search explores the entire search space below the chosen states before it backtracks on its decision. This may be inefficient if the heuristic is (locally) quite inaccurate and a wrong set of successor states is chosen, e. g. states from which no near-optimal solution can be reached.

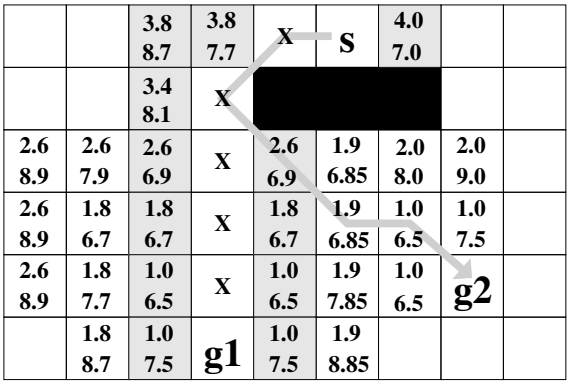
7.3 The Effect of Low-h Bias

Anytime WA* and ARA* keep the open list between search phases (possibly re-ordering it). After finding a goal state s_g, the open list will usually contain many states that are close to s_g in the search space, because the ancestors of s_g have been expanded; furthermore, those states are likely to have low heuristic values because of their proximity to s_g. Hence, if the search is continued

Initial search, $w = 2$



Continued search, $w = 1.5$



Restarted search, $w = 1.5$

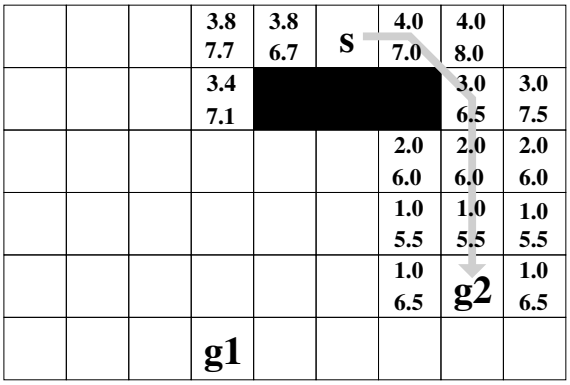


Figure 7.1: The effect of low-h bias. For all grid states generated by the search, h-values are shown above f'-values. Top: initial WA* search finds a solution of cost 6. Centre: continued search expands many states around the previous open list (grey cells), finding another sub-optimal solution of cost 6. Bottom: restarted search quickly finds the optimal solution of cost 5.

(even after updating the open list with a lower weight), it is likely to expand most of the states around s_g before considering states that are close to the initial state. This means that the search is concentrating on improving the *end* of the current solution path, as opposed to its beginning. This *low-h bias* can be a critical mistake if early decisions strongly influence final solution quality. For example in planning, it may be impossible to improve the quality of a plan substantially without changing its early actions.

Consider the example of a search problem shown in Figure 7.1. The task is to reach a goal state ($g1$ or $g2$) from the initial state s in a grid world, where the agent can move with cost 1 to each of the 8 neighbours of a cell if they are not blocked. The heuristic values are inaccurate estimates of the straight-line goal distances of cells. In particular, the heuristic values underestimate distances in the left half of the grid. We conduct a weighted A* search with weight 2 in the top panel of Figure 7.1 (assuming for simplicity a standard textbook search, i. e., no preferred operators and no deferred evaluation). Because the heuristic values to the left of s happen to be lower than to the right of s, the search expands states to the left and finds goal $g1$ with cost 6. The grey cells are generated, but not expanded in this search phase, i. e., they are in the open list. In the centre panel of Figure 7.1, the search continues with a reduced weight of 1.5. A solution with cost 5 consists in turning right from s and going to $g2$. However, the search will first expand all states in the open list that have an f'-value smaller than 7. After expanding a substantial number of states, the second solution it finds is a path which starts off left of s and takes the long way around the obstacle to $g2$, again with cost 6. If we instead *restart* with an empty open list after the first solution (bottom panel of Figure 7.1), far fewer states are expanded. The critical state to the right of s is expanded quickly and the optimal path is found.

Note that in the above example, it is in particular the systematic error of the heuristic that leads the greedy search astray and makes restarts useful. In planning, especially when using deferred evaluation, heuristic values may also be fairly inaccurate, and restarts can be useful. We will see this phenomenon again below in the experimental section of this chapter.

7.4 Restarting WA* (RWA*)

To overcome low-*h* bias, we propose Restarting WA*, or short RWA*. It runs iterated WA* with decreasing weight, always re-expanding states when it encounters a cheaper path. RWA* differs from ARA* and Anytime WA* by not keeping the open list between phases. Whenever a better solution is found, the search empties the open list and restarts from the initial state. It does, however, re-use search effort in the following way: besides the open list ("Open") and closed list ("Closed"), we keep a third data structure "Seen". When the new search phase begins, the states from the old closed list are moved to Seen. When RWA* generates a state in the new search, there are three possibilities: Case 1: The state has never been generated before (it is neither in Open nor Closed nor Seen). In this case, RWA* behaves like WA*, calculates the heuristic value of the state and inserts it into Open. Case 2: The state has been encountered in previous search phases but

```
1:  procedure RWA*({w_0, w_1, ..., w_n})
2:      bound ← ∞, w ← w_0, weight_index ← 0, Seen ← ∅
3:      while not interrupted and not failed do
4:          Closed ← ∅, Open ← {initial_state}
5:          while not interrupted and Open not empty do
6:              remove s with minimum f'(s) from Open
7:              for s' ∈ SUCCESSORS(s) do
8:                  if heuristic admissible and f(s') ≥ bound or g(s') ≥ bound then
9:                      continue
10:                 cur_g ← g(s)+ TRANSITION_COST(s, s')
11:                 if s' ∉ (Open ∪ Seen ∪ Closed) then
12:                     g(s') ← cur_g, pred(s') ← s
13:                     calculate h(s') and f'(s')
14:                     insert s' into Open
15:                 else if s' ∈ Seen then
16:                     if cur_g < g(s') then
17:                         g(s') ← cur_g, pred(s') ← s
18:                     move s' from Seen to Open
19:                 else if cur_g < g(s') then
20:                     g(s') ← cur_g, pred(s') ← s
21:                     if s' ∈ Open then
22:                         update s' in Open
23:                     else
24:                         move s' from Closed to Open
25:                 if IS_GOAL(s') then
26:                     break
27:         if new solution s* was found (line 25) then
28:             report s*
29:             bound ← g(s*)
30:             weight_index ← min(weight_index + 1, n)
31:             w ← w_weight_index
32:             Seen ← Seen ∪ Open ∪ Closed
33:         else
34:             return failure
```

Algorithm 7.1: Pseudo-code for the RWA* algorithm.

not in the current phase (it is in Seen). In this case, RWA* can look up the heuristic value of the state rather than having to calculate it. In addition, RWA* checks whether it has found a cheaper path to the state or whether the previously found path is better, and keeps the better one. The state is removed from Seen and put into Open. Case 3: The state has already been encountered in this search phase (it is in Open or Closed). In this case, RWA* again behaves like WA*, re-inserting the state into Open only if it has found a cheaper path to the state. Complete pseudo-code is shown in Figure 7.1. This strategy can be implemented in an efficient way by maintaining a boolean value "seen" in each state of the open/closed list, rather than keeping the seen states in a separate list.

In short, previous search effort is re-used by not calculating the heuristic value of a state more than once and by making use of the best path to a state found so far. Compared to ARA* and Anytime A*, our method may have to re-expand many states that were already expanded in previous phases. However, in cases where the calculation of the heuristic accounts for the largest part of computation time (as in our planning experiments), the duplicated expansions do not have much effect on runtime. Thus, RWA* re-uses most of the previous search effort, but its restarts allow for more flexibility in discovering different solutions.

7.4.1 Restarts in Other Search Paradigms

Restarts are a well-known and successful technique in combinatorial search and optimisation, e. g. in the areas of propositional satisfiability and constraint-satisfaction problems. Together with randomisation, they are used in systematic search (Gomes et al., 1998) as well as local search (Selman et al., 1992) to escape from barren areas of the search space. A restart is typically executed if no solution has been found after a certain number of steps. In most of these approaches restarting would be useless without randomisation, as the algorithm would behave exactly the same each time. By contrast, our algorithm is deterministic, and its restarts serve the purpose of revisiting nodes near the start of the search tree to promote exploration when the node evaluation function has changed.

This is perhaps most closely related to the motivation behind limited-discrepancy search (LDS) (Harvey and Ginsberg, 1995; Furcy and Koenig, 2005). LDS attempts to overcome the tendency of depth-first search to visit solutions that differ only in the last few steps. It backtracks whenever the current solution path exceeds a given limit on the number of discrepancies from the greedy path, restarting from the root each time this limit is increased. Like random restarts and the backtracking in limited-discrepancy search, our restarts mitigate the effect of bad heuristic recommendations early in the search. Our use of restarts differs from limited-discrepancy backtracking in that we restart upon finding a solution, rather than after expanding all nodes with a given number of discrepancies from the greedy path. As opposed to random restarts, we restart upon success (when finding a new solution) rather than upon failure (when no solution could be found). A depth-first approach like LDS is however not competitive for typical planning tasks.

Lastly, incomplete methods can be made complete by restarting with increasingly relaxed heuristic restrictions (Zhang, 1998; Aine et al., 2007). By contrast, our restarts occur within an

algorithm that is already complete.

Our use of restarts also raises an interesting connection between anytime search in deterministic and stochastic domains. An analogy can be made between RWA* and real-time dynamic programming (RTDP), a popular anytime algorithm for solving Markov decision processes (Barto et al., 1995). Both algorithms follow a greedy strategy relative to the current state values until they find a goal, at which point they return to the initial state for another trial. In RTDP, Bellman back-ups are performed to increase the accuracy of heuristic values, while in RWA*, the weight is decreased for a similar effect. The procedures do have significant differences: RTDP makes irrevocable moves in the style of a local search, while RWA* keeps an open list. RTDP requires a labelling procedure to detect optimality (Bonet and Geffner, 2003), while RWA* can terminate after a trial with weight 1 (when being used with an admissible heuristic and without any modifications such as search enhancements; otherwise it can be extended easily to prove optimality by exhausting those states in Open that have a lower g-value than the incumbent solution). It would be an interesting topic for future work to investigate whether importing ideas from RTDP, such as backing up heuristic values, could benefit the anytime profile of RWA*.

Our use of restarts in a complete best-first search is remarkable because at first glance, it would seem unnecessary: a best-first search keeps a queue of all generated but yet unexpanded nodes, and is thus not thought to suffer from premature commitment. Our finding is that a greedy search like weighted A* effectively makes such commitments due to its low-h bias.

7.5 Empirical Evaluation

We compare the performance of RWA* with several existing complete anytime approaches: Anytime A* (Hansen and Zhou, 2007), ARA* (Likhachev et al., 2004), Anytime Window A* (Aine et al., 2007), and beam-stack search (Zhou and Hansen, 2005). For beam-stack search, we implemented the regular version rather than the memory-conserving divide-and-conquer variant (since memory-conserving techniques could be applied to all the algorithms we compare). All parameters of the competitor algorithms were carefully tuned for best performance. The algorithms based on WA* (RWA*, Anytime A* and ARA*) all share the same code base, as they differ only in few details; and they all perform best for the same starting weights. For beam-stack search, we experimented with beam width values between 5 and 10,000 and plot the best of those runs. We also conducted experiments with iteratively broadening beam widths, but did not obtain better results than with the best fixed value. Beam-stack search expects an initial upper bound on the solution cost. After consulting its authors, we chose a safe but large number as bound, as it is not clear how to come up with a reasonable upper bound for our experiments up-front.

In addition we compare against an alternative version of Anytime A*. For this algorithm, which originally does not decrease its weight between search phases, we experienced improved performance if we do decrease the weight. Thus, we also report results for a variant dubbed "Anytime A* WS", (WS for weight schedule), where the weight is decreased as in RWA* or

ARA*.

7.5.1 Automated Planning

We implemented each of the search algorithms within the Fast Downward planning framework (Helmert, 2006), replacing the greedy best-first search in Fast Downward with the respective search algorithms. We mainly use the FF/add heuristic for the results reported here, but we also tested other heuristics and briefly report on them. All other settings of Fast Downward were held fixed to isolate the effect of changing the search mechanism. In particular, since we want to achieve good performance we make use of the search enhancements *preferred operators* and *deferred heuristic evaluation* used by Fast Downward (see Section 2.2.3). For all A*-based algorithms, the preferred-operator mechanism can be used in the same way as in Fast Downward's greedy best-first search and it improves results notably; it was thus incorporated. Beam-stack search is the one algorithm that does not use preferred operators as it is not obvious how to incorporate them.

For the WA*-based algorithms, we experimented with several starting weights and corresponding weight sequences, yet found that the relative performance of the algorithms did not change much. The weight sequence that gave overall best results and is used in the results below is 5, 3, 2, 1.5, 1. ARA* was adapted to use a weight schedule like RWA* and Anytime A* WS here, since the original mechanism for decreasing weights in ARA* depends on the heuristic being admissible, whereas the heuristics we use in satisficing planning are inadmissible. (ARA* originally reduces its weight by a very small amount whenever it proves that the incumbent solution is w-admissible for the current weight w.) For beam-stack search, a beam width of 500 resulted in best anytime performance.

We use all classical planning tasks from the International Planning Competitions between 1998 and 2006 except for the trivial Movie domain. The time and memory limits were 30 minutes and 3 GB respectively for each task, running on a 3 GHz Intel Xeon X5450 CPU.

Results. Figure 7.2 summarises the results. We plot average IPC scores following the definition used in the International Planning Competition 2008 (see Section 2.4): for each task from a given domain, an algorithm scores c^*/c at time t, where c is the cost of the algorithm's current solution at time t and c^* is the cost of the overall best solution found by any of the algorithms at the end of the timeout. These scores are then averaged over all tasks from a domain and over all domains, so that each domain has equal influence on the final score irrespective of its size. To support the visual differences with statistical analysis, we give results using the Wilcoxon signed-rank test, a statistical hypothesis test for two related samples that does not rely on the assumption that the data samples are drawn from a probability distribution of any particular form.

Legends in our plots are sorted in decreasing order of final score. As the top and centre panels of Figure 7.2 show, RWA* performs best, outperforming the other algorithms. In the top panel, we compare against the other WA*-based algorithms. The weight-decreasing variant of Anytime A* (Anytime A* WS) is slightly better than ARA*, whereas the original Anytime A*

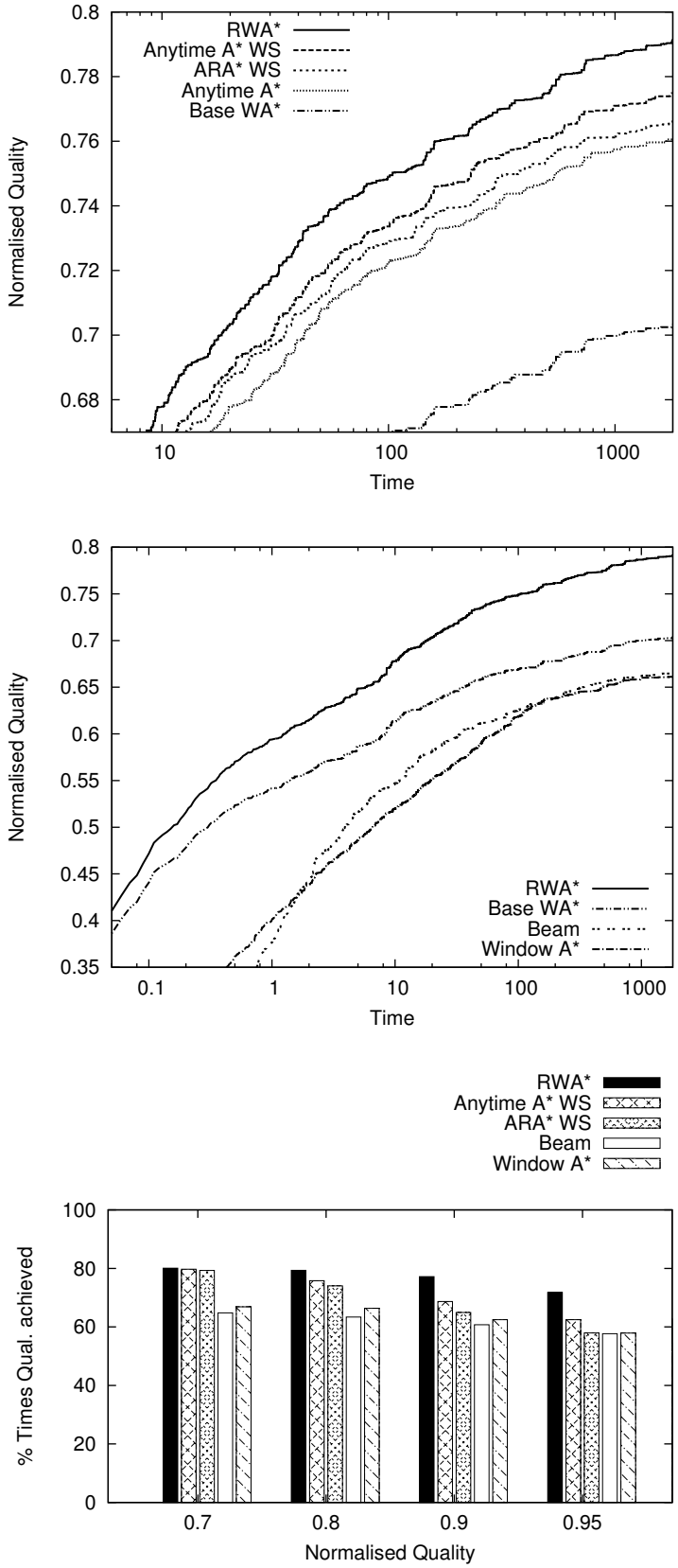


Figure 7.2: Anytime performance in planning.

algorithm performs worse. It is notable that while all four WA*-based algorithms are very closely related, differing only in a few lines of code, the simple addition of the restarts leads to a notable improvement in performance. "Base WA*" depicts a WA* search that stops after finding the first solution and is provided for reference.[1]

Wilcoxon signed-rank tests conducted at several points along the curves reveal that the differences between all pairs of algorithms are statistically significant except for the difference between ARA* and the original Anytime A* algorithm. The p-values for the significant differences are $\leq$ 0.001.

The centre panel shows the results for Window A* and beam-stack search. Beam-stack search and Window A* perform significantly worse than the WA*-based algorithms and worse than the WA* baseline that does not do anytime search. This is mainly due to solving fewer tasks, as a significant determiner in gross performance is the number of tasks solved by a certain time. By the time cut-off, the WA*-based algorithms leave (essentially the same) 256–259 tasks unsolved, while beam-stack search and Window A* fail to solve 347 and 591 tasks, respectively. Beam-stack search (Window A*) solves 54 (46) tasks that RWA* does not solve. The opposite is true for 145 (381) tasks. While Window A*solves significantly fewer tasks than beam-stack search, it produces higher-quality solutions, and the resulting performance curves are very similar for the two algorithms.

The bottom panel of Figure 7.2 shows for the best version of each algorithm how often it achieved (at least) a certain solution quality. As can be seen, RWA* achieves high-quality solutions more often than the other algorithms. The best quality is achieved by RWA* 63% of the time, (where domains are weighed for equal influence as above), while all others achieve it $\leq$ 55% of the time. The average number of solutions found was 1.9–2 for the weight-decreasing WA* methods and beam-stack search, 2.7 for non-decreasing Anytime A* and 1.2 for Window A*.

When examining the distribution of RWA*'s performance over the different domains, we found that RWA* outperforms the other WA* algorithms in 40% of the domains, while being on par in the remaining 60%. In no domain did RWA* perform notably worse than any of the other WA* methods. Beam-stack search and Window A* show different strengths and weaknesses compared to the WA*-based algorithms. In the Optical Telegraph domain, beam-stack search performs well whereas all other algorithms perform badly. Beam-stack search and Window A* fare worst on large tasks, e. g. in the Schedule and Logistics 1998 domains.

To summarise, we find that the WA*-based methods achieve significantly better results than Window A* and beam-stack search. We believe that this is due to the *global* WA*-based algorithms being better able to deal with inadmissible and locally highly varying heuristic values, whereas Window A* and beam-stack search commit early to certain areas of the search space. Exhausting

[1]The difference to the results in Chapter 6 is due to a number of factors, including code evolution and differences in the machines used for the experimental evaluation. However, a major factor is that the IPC score is heavily dependent on the best known solution for each task (see Section 2.4). In this experiment, we compare a range of varied algorithms with different strengths. This may lead to near-optimal solutions being produced more often here than in the experiment of Chapter 6, resulting in overall lower scores.

	1st solution				2nd solution				3rd (final) solution			
	len	*c.i.*	*exp*	*t*	*len*	*c.i.*	*exp*	*t*	*len*	*c.i.*	*exp*	*t*
RWA*	165	-	1142	0.07	163	153	2370	0.08	**125**	**1**	16919	0.84
ARA*	165	-	1142	0.06	163	153	1220	0.07	161	145	5743	0.28
Anyt. A* WS	165	-	1142	0.07	163	153	1231	0.08	161	145	5629	13.25
Anyt. A*	165	-	1142	0.07	163	153	1255	0.07	161	145	206480	0.28

Table 7.1: Solution sequences for the largest task in the Gripper domain. The plan length is denoted by *len*. The first step in which a solution deviates from the previous is denoted by *c.i.* (change index); *exp* is the number of states expanded until the solution is found, and *t* the runtime in seconds.

a state-space area is particularly difficult here as inadmissible heuristic values cannot be used for pruning. Furthermore, beam-stack search cannot make use of the planning-specific preferred operators enhancement.

We also conducted experiments without preferred operators and delayed evaluation for all algorithms. There, the performance of all algorithms becomes drastically worse (e. g., the WA*-based methods leave twice as many problems unsolved and in addition show worse anytime performance on the solved problems). RWA* still performs better than all other algorithms, though by a much smaller margin than with the search enhancements. With fewer solved tasks and fewer solutions per task, RWA* cannot improve on the other WA* algorithms as much. Also, while the search enhancements are very helpful in finding a solution, they make the search slightly less informed and more greedy, contributing to low-*h* bias and thus making restarts more effective.

We furthermore conducted experiments with other planning heuristics (not shown), namely the landmark heuristic from Chapter 4, the additive heuristic, the causal graph heuristic and the context-enhanced additive heuristic. Compared to the experiments described above, which use the FF/add heuristic, the anytime performance profiles of all algorithms were worse with these other heuristics. However, the *relative* performance of the algorithms was similar, with RWA* outperforming the other methods in all cases except with the causal graph heuristic (where the weight-decreasing WA* algorithms were on par). The differences between the various algorithms however were smaller with these other heuristics. This can be explained by the fact that worse performance of the algorithms results in their anytime profile curves being compressed, thus decreasing the differences between them.

A detailed example. We argue that RWA* shows such good results because restarts encourage changes in the beginning of a plan rather than the end. The largest task in the Gripper domain (see Section 2.4) provides an illustration. The initial WA* search phase finds a plan in which all balls are transported separately, whereas in the optimal plan the robot always carries two balls at a time. Window A* does not solve this task, while beam-stack search finds only one solution (the optimal) after 2.5 seconds. The WA* algorithms find three improving solutions, with the first solution found after less than one second, see Table 7.1. All of these plans are found very quickly, in less than 15 seconds. However, the last plan found by RWA* is optimal, whereas the other WA* algorithms do not find any improved solutions during the remaining 29 minutes.

The noteworthy aspect in this example is the *indices of change* for the plans, i. e. the steps in which subsequent plans first differ from their predecessors. For ARA* and the two Anytime A* variants these change indices are fairly high (≥ 145), denoting the fact that their subsequent plans only differ in the last 20 actions from the first plan found. For RWA*, the third solution has a change index of 1, i. e. it differs in the first action from the previous plans. This suggests that the big jump in solution quality for RWA* indeed results from further exploration near the initial state, whereas the other algorithms unsuccessfully spend their effort in deeper areas of the search space.

7.5.2 Other Benchmarks

To see whether restarts in anytime search are also beneficial in problems outside planning, we conducted experiments in three further benchmark domains: the Robotic Arm domain, the Grid World path-finding domain, and the Sliding-Tile Puzzle. We chose these domains because of their previous use as anytime benchmarks (Robotic Arm), or their standing as traditional benchmarks in the search literature (Grid World and the Sliding-Tile Puzzle).

Robotic Arm. This domain concerns motion planning for a simulated 20-degree-of-freedom robotic arm (Likhachev et al., 2004). The base of the arm is fixed, and the task is to move its end-effector to the goal while navigating around obstacles. An action is defined as a change of the global angle at a particular joint. The workspace is discretised by overlaying it with a grid of 50×50 cells, and the heuristic of a state is the distance from the current location of the robot's end-effector to the goal, taking obstacles into account. The size of the state space is over 10^{26}, and in most instances it is infeasible to find an optimal solution. We experiment with 22 tasks kindly made available to us by Maxim Likhachev.

For ARA*, we use the settings suggested by Likhachev et al., starting with a weight of 3 and decreasing it by 0.02 each time; the other WA*-based algorithms also start with weight 3. Note that because the heuristic is admissible, ARA* can prove suboptimality bounds and make informed decisions on when to reduce its weight (namely, whenever a new bound is proven). For RWA*, on the other hand, a decrease in weight incurs a substantial overhead each time due to the restart. This is why larger and less frequent weight decreases work better for it. We use a factor of 0.84 to decay the weight for RWA*, resulting in a similar weight schedule as in the planning experiment, reducing the weight whenever an improved solution is found.

The results are shown in the top panel of Figure 7.3. RWA* outperforms ARA* and Anytime A* for all timeouts above 1 second and shows overall very good anytime performance, including reaching the best final quality. Window A* does well in this domain and shows best results in the early stages of search. Anytime A* performs notably worse than ARA* here, though we found that its weight-decreasing variant (not shown), decreasing the weight in the same way ARA* does, achieves similar performance as ARA*. Beam-stack search, here with a beam width of 100, takes longer than most of the other methods to achieve high performance, but then comes up steeply and achieves almost the same final quality as RWA*. RWA* finds 3.7 different solutions per instance

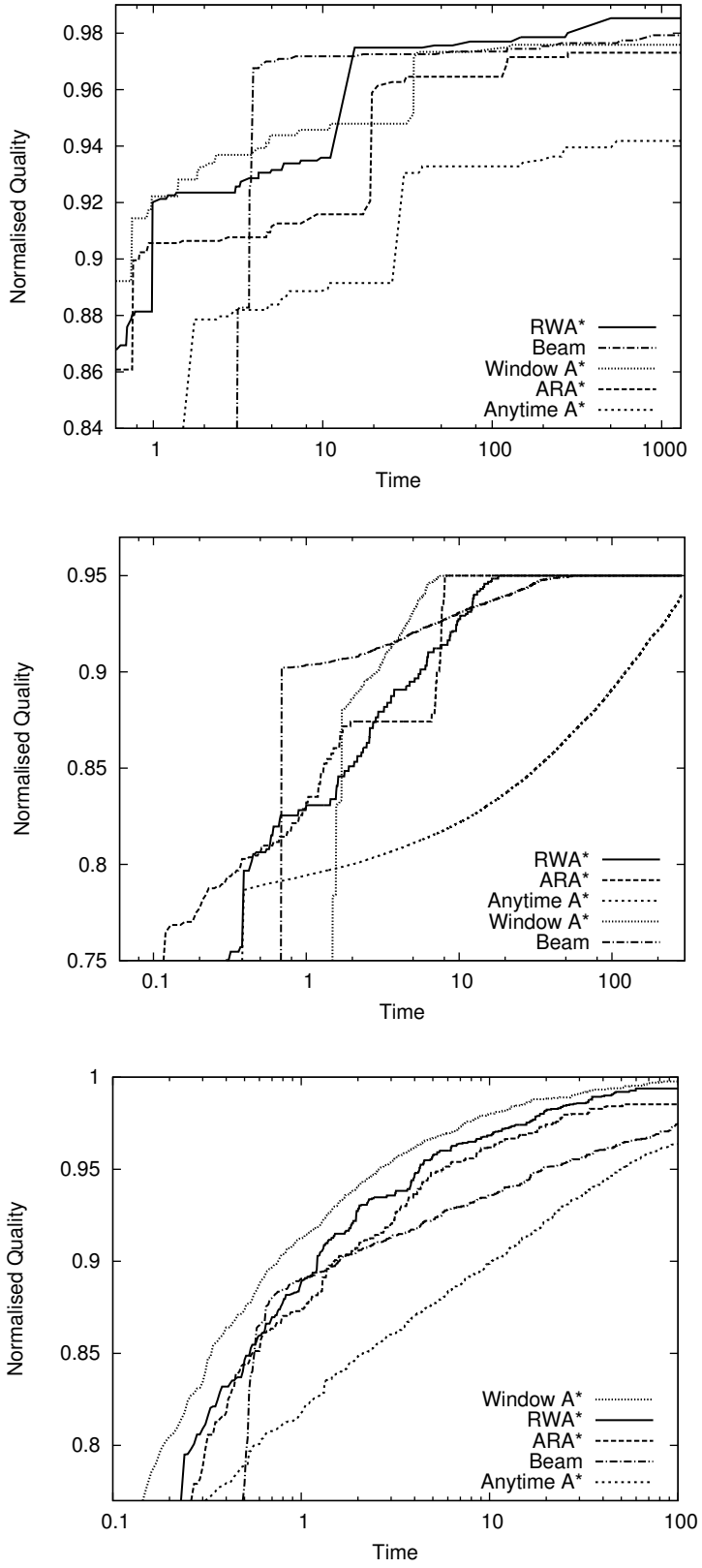


Figure 7.3: Anytime performance for the Robot Arm domain (top), Grid World (middle), and the Sliding-Tile Puzzle (bottom).

on average while Anytime A* finds 3.0 and the other algorithms find fewer than 2.7.

On these hand-designed instances, the variance in performance is fairly high. According to Wilcoxon signed-rank tests, Anytime A* is significantly worse than the other algorithms, with $p \leq 0.001$, and RWA* is significantly better than ARA*, with $p \leq 0.05$. Furthermore, the final quality of RWA* is significantly better than that of Window A*, with $p \leq 0.05$. For all other cases, results are not conclusive in the second half of the plot, where RWA*, ARA*, Window A* and beam-stack search are very close.

Grid World. Grid world tasks consist in finding a path from the top left to the bottom right in a 2000×1200 cell grid, using 4-way movement. We randomly generated 20 tasks with a probability of 0.35 of each cell being blocked. We used a unit cost function and a Manhattan distance heuristic which ignores obstacles. In this domain, relatively low weights gave best results for the WA*-based algorithms. The best starting weight was 2, with the weight schedule of RWA* being 2, 1.5, 1.25, 1.125, while ARA* uses small weight decrements of 0.2. Beam-stack search used a beam width of 50.

Results are shown in the centre panel of Figure 7.3. The tasks are solved optimally within 60 seconds by all algorithms except non-decreasing Anytime A*. RWA* and ARA* show the best anytime performance and perform very similarly. ARA* performs better than the others in the first split second, beam-stack search between seconds 0.8–3. However, beam-stack search and Window A* take longer than RWA* and ARA* to achieve good quality. Anytime A* did not perform well, though we found again that its weight-decreasing variant (not shown) performs similarly to RWA* and ARA*. With smaller beam widths, the score of beam-stack search rises faster in the beginning but it takes longer to achieve perfect quality; with larger beam widths the reverse holds. With beam width 100, for example, beam-stack search performs similarly to Window A* in this domain.

Wilcoxon signed-rank tests confirm that the differences between the algorithms in the time frame between 2 and 8 seconds are statistically significant with $p \leq 0.001$, except close to the cross-over points of two curves.

Sliding-Tile Puzzle. In the Sliding-Tile Puzzle domain, we tested on the 100 15-puzzle instances from Korf (1985) using the Manhattan distance heuristic. The best starting weight for the WA* algorithms was 3, the weight sequence for RWA* being 3, 2, 1.5, 1.25, 1 while ARA* uses small decrements of 0.2. The beam width in beam-stack search was 500.

The results are shown in the bottom panel of Figure 7.3. There are comparatively few different solutions, leading to similar performance of all WA*-based algorithms that decrease weight. Window A* performs best, with RWA* second and ARA* third. Beam-stack search performs less well, showing the same behaviour relative to its beam width as in the Grid World domain. Non-decreasing Anytime A* performs worst, though we found again that its weight-decreasing variant (not shown) performs similarly to RWA* and ARA*.

According to Wilcoxon signed-rank tests, Window A* is significantly better than RWA* be-

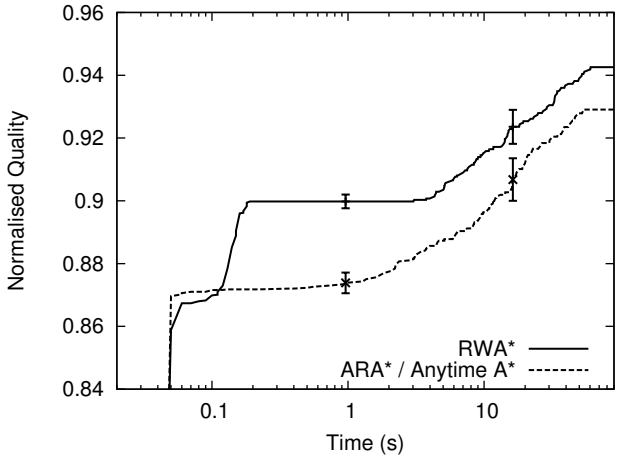


Figure 7.4: Anytime performance in an artificial search space.

tween seconds 2 and 20 with $p \leq 0.05$, but the differences after 20 seconds are not statistically significant. The differences between all other algorithms are significant with at least $p \leq 0.05$.

Summary. While in the Robotic Arm domain, Grid World, and the Sliding-Tile Puzzle RWA* does not dominate its competitors as notably as in the planning experiment, we note that its performance is continually very good. Compared to its most similar competitors, the other WA* algorithms, RWA* is always better (Robotic Arm) or on par (Grid World, Sliding-Tile Puzzle). This may suggest that even in cases where the restarts are not helpful, they do little harm in terms of computation time. The methods that are not based on WA* (Window A* and beam-stack search) perform well in some domains but badly in others. By contrast, RWA* shows robustly good performance over all domains.

7.5.3 An Artificial Search Space

To get a clearer picture of when restarts are helpful, we conducted experiments on a manually-designed search tree similar to the one used in Chapter 6. We fixed a branching factor of 25 and a typical solution depth of 500. Nodes are characterised by their *approximate goal distance (agd)*. The root has an *agd*-value of 500 and nodes with a value of 0 are labelled as goals. Edge costs are chosen randomly (with uniform probabilities) between 0 and 10, and for a given edge of cost c the *agd*-value of the reached child varies randomly by up to c from the parent's value. This is analogous to the costs of walking in a Grid World, where a move of c steps may take an agent c steps closer to the goal, c steps further away or anything in between, depending on the direction the agent walks.

The heuristic values were chosen to enforce systematic biases. The heuristic underestimates the *agd*-values of nodes by up to a certain percentage. The heuristic value of the root is chosen randomly with this constraint, and the resulting error factor (the ratio of heuristic value to *agd*-value) is recorded. For all other nodes, their heuristic values are correlated with their parent's

heuristic values such that a child's value differs by no more than 1 from the value that would result when using the same error factor as the parent. This is achieved by first computing the "parent-induced value" that would result if a node had the same error factor as its parent. Then we determine a random value for the child within the constraint of heuristic accuracy. The actual value for the child is obtained by moving the parent-induced value h_p by m into the direction of the random value h_r, where $m = \max(1, |h_p - h_r|)$.

Averaged results for 100 runs with different random seeds for a heuristic that underestimates by up to 20% are shown in Figure 7.4, where RWA* and ARA* were run with a weight schedule of 2, 1.5, 1.25, 1.125, 1. The relative results are similar for heuristics of different accuracy. Note that the weight-decreasing variant of Anytime A* is equivalent to ARA* here, as there are no cycles in the search space. RWA* has a clear advantage over the other algorithms, and we found the same effect as in planning, that the change indices of solutions for RWA* are substantially lower than for the other algorithms.

When weakening the correlation between the heuristic values of parents and children, the advantage of RWA* diminishes. For example, if all heuristic values are chosen completely at random, RWA* has no advantage over ARA*/Anytime A* unless a very accurate heuristic is used. This is due to the fact that without systematic errors in the heuristic, all algorithms explore more nodes in the top of the search tree, rather than committing quickly to a bad path. This is in line with observations by Zahavi et al. (2007), who note that inconsistent heuristics can be beneficial because they make it less likely that the search gets stuck in a region of bad heuristic estimates. In that case, the greedy search makes fewer early mistakes and restarts consequently provide less benefit.

Additionally, we found that RWA* gains more advantage if the heuristic underestimates nodes near the root by a stronger factor than nodes that are lower in the tree. In none of the scenarios we looked at did RWA* perform notably worse than the other algorithms.

7.6 Discussion

As we saw in the example tasks from the Grid World and the Gripper domain, restarting anytime search overcomes the low-h bias of greedy anytime search that tends to expand nodes near a known goal. The desire to revisit nodes near the initial state stems from the fact that a heuristic evaluation function is often less informed near the root of the search tree than near a goal and that early mistakes can be important to correct (Harvey and Ginsberg, 1995; Furcy and Koenig, 2005). For example, in tasks with multiple goal states the heuristic may misjudge the relative distance of the goals and lead a greedy search into a suboptimal part of the search space. Similarly, search with inaccurate heuristics that have systematic errors can lead to early mistakes and thus benefit from restarts.

7.7 Conclusion

We have demonstrated an important dysfunction in conventional approaches to anytime search: by trying to re-use previous search effort from a greedy search, traditional anytime methods suffer from low-h bias and tend to improve the incumbent solution starting from the end of the solution path. As we have shown, this can be a poor strategy when the heuristic makes early mistakes. The counter-intuitive approach of clearing the open list, as in the Restarting Weighted A* algorithm, can lead to much better performance in such domains while doing little harm in others.

8

The LAMA Planner

In this chapter, we integrate the most successful ideas for satisficing planning from our previous chapters. The resulting planning system LAMA showed best performance among all competing planners in the sequential satisficing track of the International Planning Competition in 2008. LAMA uses landmarks in addition to the FF/add heuristic, and employs the search enhancements *preferred operators* and *deferred heuristic evaluation* in their most effective combination. It finds a first solution quickly using greedy best-first search and then improves on this solution using restarting anytime search as long as time remains.

In this chapter, we present the LAMA planning system in detail and investigate which of its distinguishing features are crucial for its good performance and how these features interact. We find that the landmark heuristic and the restarting anytime search both contribute to improving the results, and there are even synergy effects between them. While the incorporation of action costs into the heuristic estimators, as discussed in Chapter 5, was not beneficial given the IPC 2008 benchmarks, the landmarks mitigate the bad performance of the cost-sensitive FF/add heuristic in LAMA. We also show results on traditional benchmarks without action costs, demonstrating that LAMA is competitive with the state of the art in satisficing unit-cost planning if we are interested in good solution quality.

8.1 Introduction

LAMA builds on the implementations discussed in previous chapters of this thesis, which in turn are based on the Fast Downward planning system (Helmert, 2006). LAMA differs from Fast Downward in three major ways:

1. **Landmarks.** In LAMA, Fast Downward's causal graph heuristic is replaced with the combination of the FF/add heuristic and the landmark heuristic (see Section 2.2.2 and Chapter 4, respectively). As shown in Chapters 4 and 5, this combination leads to better coverage and higher-quality solutions than using only the FF/add heuristic or the causal graph heuristic.

2. **Action costs.** Both the landmark heuristic and the FF/add heuristic use action costs as described in Chapter 5. However, LAMA does not focus purely on the *cost-to-go*, i. e., the

estimated cost of reaching the goal from a given state. There is a danger that a cost-sensitive planner may concentrate too much on finding a cheap plan, at the expense of finding a plan at all within a given time limit. LAMA weighs the estimated cost-to-go (as a measure of plan quality) against the estimated goal distance (as a measure of remaining search effort) by combining the values for the two estimates.

3. **Anytime search.** LAMA continues to search for better solutions until it has exhausted the search space or is interrupted. After finding an initial solution with a greedy best-first search, it conducts a series of weighted A* searches with decreasing weights, restarting the search each time from the initial state when an improved solution is found. As discussed in Chapter 7, we found this approach to be very efficient on planning benchmarks compared to other anytime methods.

As shown in Chapter 5, the use of action costs in the FF/add heuristic proved not to be beneficial on the IPC 2008 benchmarks. However, this result was not expected at the time of LAMA's creation, so that LAMA uses the cost-sensitive versions of its heuristics, aimed at finding high-quality plans. Despite the slightly detrimental impact of its cost-awareness, however, LAMA outperformed its competitors by a substantial margin at the International Planning Competition 2008. In particular, all other cost-sensitive planners in the competition fared worse than the baseline planner FF that ignored costs, demonstrating that cost-based planning presents a considerable challenge (as discussed in detail in Chapter 5).

This chapter provides an evaluation of LAMA's performance in the setting provided by IPC 2008. We aim to elicit how much the performance of the LAMA system as a whole is influenced by each of the three distinguishing features described above (landmarks, action costs and anytime search). To answer this question, we contrast several variations of our planner using various subsets of these features. As discussed already in Chapter 5, the use of landmarks effectively mitigates the problems of the cost-sensitive FF/add heuristic. The combination of cost and distance estimates in LAMA proves to be beneficial when using iterated search. The anytime search significantly improves the quality of solutions throughout and even acts in synergy with landmarks in one domain.

8.2 System Architecture

LAMA inherits the overall structure and large parts of its functionality from Fast Downward (Helmert, 2006). Like Fast Downward, LAMA accepts input in the PDDL2.2 Level 1 format (Fox and Long, 2003; Edelkamp and Hoffmann, 2004), including ADL conditions and effects and derived predicates. In addition, we have implemented the functionality to handle the action costs introduced for IPC 2008 (Helmert et al., 2008). Like Fast Downward, LAMA consists of three separate components:

- The translation module

- The knowledge compilation module
- The search module

These components are implemented as separate programs that are invoked in sequence. In the following, we provide a brief description of the translation and knowledge compilation modules. The main changes in LAMA, compared to Fast Downward, are implemented in the search module, which we discuss in more detail.

8.2.1 Translation

The *translation module*, short *translator*, transforms the PDDL input into a planning task in finite-domain representation as described in Section 2.1. The main components of the translator are an efficient grounding algorithm for instantiating schematic operators and axioms, and an invariant synthesis algorithm for determining groups of mutually exclusive facts. Such fact groups are consequently replaced by a single state variable, encoding *which* fact (if any) from the group is satisfied in a given world state. Details on this component can be found in a recent article by Helmert (2009).

The groups of mutually exclusive facts (mutexes) found during translation are later used to determine orderings between landmarks. For this reason, LAMA does not use the finite-domain representations offered at IPC 2008 (object fluents), but instead performs its own translation from binary to finite-domain variables. While not all mutexes computed by the translation module are needed for the new encoding of the planning task, the module has been extended in LAMA to retain all found mutexes for their later use with landmarks.

Further changes we made, compared to the translation module described by Helmert, were to add the capability of handling action costs, implement an extension concerning the parsing of complex operator effect formulas, and limit the runtime of the invariant synthesis algorithm. As invariant synthesis may be time critical, in particular on large (grounded) PDDL input, we limit the maximum number of considered mutex candidates in the algorithm, and abort it, if necessary, after five minutes. Note that finding few or no mutexes does not change the way the translation module works; if no mutexes are found, the resulting encoding of the planning task contains simply the same (binary-range) state variables as the PDDL input. When analysing the competition results, we find that the synthesis algorithm aborts only in some of the tasks of one domain (Cyber Security).

8.2.2 Knowledge Compilation

Using the finite-domain representation generated by the translator, the *knowledge compilation* module is responsible for building a number of data structures that play a central role in the subsequent landmark detection and search. Firstly, *domain transition graphs* (see Definition 2.3) are produced which encode the ways in which each state variable may change its value through action

applications and axioms. Furthermore, data structures are constructed for efficiently determining the set of applicable actions in a state and for evaluating the values of derived state variables. We refer to Helmert (2006) for more detail on the knowledge compilation component, which LAMA inherits unchanged from Fast Downward.

8.2.3 Search

The search module is responsible for the actual planning. Two algorithms for heuristic search are implemented in LAMA: (a) a greedy best-first search, aimed at finding a solution as quickly as possible, and (b) a weighted A* search that allows balancing speed against solution quality.

Both search algorithms use the three search enhancements *multi-heuristic search*, *preferred operators* and *deferred heuristic evaluation* (see Section 2.2.3) that were inherited from Fast Downward and proven useful in Chapters 4 and 6.

Restarting Anytime Search

LAMA is a cost-based planner that aims to deliver high-quality solutions. Guiding the search towards cheap goals is achieved in two ways in LAMA: firstly, both the landmark heuristic and the FF/add heuristic in LAMA are capable of handling action costs and estimating the cost-to-go, in addition to the distance-to-go, from a given state. Secondly, the search algorithm does not only take the cost-to-go from a given state into account, but also the cost necessary for reaching that state. This is the case for weighted A* search as used in LAMA.

To make the most of any available time, LAMA employs an anytime approach similar to the one discussed in Chapter 7: it first runs a greedy best-first search, aimed at finding a solution as quickly as possible. Once a plan is found, it searches for progressively better solutions by running a series of weighted A* searches with decreasing weight. The cost of the best known solution is used for pruning the search, while decreasing the weight over time makes the search progressively less greedy, trading speed for solution quality. Whenever a new solution is found, LAMA starts a new weighted A* search, discarding the open lists of the previous search and re-starting from the initial state. While resulting in some duplicate effort, these restarts can help overcome bad decisions made by the early (comparatively greedy) search iterations with high weight (see Chapter 7). In contrast to the approach presented in Chapter 7, however, LAMA uses the simpler method of conducting a series of entirely independent searches, not retaining any heuristic values or g-values between searches. The reason for this is that we added the value caching to our restarting approach *after* LAMA's implementation and participation at IPC 2008. In this chapter, we want to evaluate the LAMA planner that competed at IPC 2008, so that we do not include the caching mechanism. We found that even using the simpler method of restarting without caching, the restarts still provide considerable improvement over related anytime approaches that do not restart.

Using Cost and Distance Estimates

Both heuristic functions used in LAMA are cost-sensitive, aiming to guide the search towards high-quality solutions. Focusing a planner purely on action costs, however, may be dangerous, as cheap plans may be longer and more difficult to find, which in the worst case could mean that the planner fails to find a plan at all within the given time limit. Zero-cost actions present a particular challenge: since zero-cost actions can always be added to a search path "for free", even a cost-sensitive search algorithm like weighted A* may explore very long search paths without getting closer to a goal. In Chapter 5, we broke ties in the cost-sensitive search according to distance estimates in order to counter-act the problem associated with zero-cost actions. However, this method still has a very strong emphasis on costs.

Methods have been suggested that allow a trade-off between the putative cost-to-go and the estimated goal distance (Gerevini and Serina, 2002; Ruml and Do, 2007). However, they require the user to specify the relative importance of cost versus distance up-front, a choice that was not obvious in the context of IPC 2008. LAMA gives equal weight to the cost and distance estimates by adding the two values during the computation of its heuristic functions. For the FF/add heuristic, this means that during cost propagation, each action contributes its action cost *plus 1 for its distance*, rather than just its action cost, to the propagated cost estimates. Analogously, the landmark heuristic sums cost and distance estimates by adding 1 to the estimated cost of each landmark.

This measure of balancing cost and distance estimates in LAMA is a very simple one, and its effect changes depending on the magnitude and variation of action costs in a task: the smaller action costs are, the more this method favours short plans over cheap plans. For example, 5 zero-cost actions result in an estimated cost of 5, whereas 2 actions of cost 1 result in an estimated cost of 4. LAMA would thus prefer the 2 actions of cost 1 over the 5 zero-cost actions. By contrast, when the action costs in a planning task are larger than the length of typical plans, the cost estimates dominate the distance estimates and LAMA is completely guided by costs. Nevertheless this simple measure proves useful on the IPC 2008 benchmarks, outperforming pure cost search (with tie-breaking) in our experiments. More sophisticated methods of automatically balancing cost against distance (for example by normalising the action costs in a given task with respect to their mean or median) are a potential topic of future work.

8.3 Experimental Evaluation

To evaluate how much each of the central features of LAMA contributes to its performance, we have conducted a number of experiments comparing different configurations of these features. We focus our detailed evaluation on the benchmark tasks from the International Planning Competition 2008, as we are interested in the setting of planning with action costs. The effect of landmarks in classical planning tasks without actions costs has been studied in Chapter 4, but we provide summarising results for this case, using the domains of IPCs 1998–2006, in Section 8.3.6.

As mentioned before, LAMA differs from Fast Downward in three major ways: (1) through the use of landmarks, (2) by using cost-sensitive heuristics and combining cost and distance estimates to balance speed and quality of the search, and (3) by employing anytime search to continue to search for better solutions while time remains. To measure the benefit of LAMA's approach to costs, we test three different methods of dealing with costs: (a) using the traditional cost-unaware heuristics (distance estimates), (b) using purely cost-sensitive heuristics (though using distance estimates for tie-breaking), and (c) using the combination of the distance and cost estimates, as in LAMA. The different choices regarding landmarks and approaches to action costs then result in the following six planner configurations:

- **F**: Use the cost-unaware FF/add heuristic (estimating goal distance).
- $\mathbf{F_c}$: Use the purely cost-sensitive FF/add heuristic (estimating cost-to-go).
- $\mathbf{F_c^+}$: Use the FF/add heuristic that combines action costs and distances.
- **FL**: Use the cost-unaware variants of both the FF/add heuristic and the landmark heuristic.
- $\mathbf{FL_c}$: Use the purely cost-sensitive variants of both heuristics.
- $\mathbf{FL_c^+}$: Use the variants that combine action costs and distances for both heuristics.

Each configuration is run with iterated (anytime) search. When highlighting the contribution of the iterated search, we report first solutions vs. final solutions, where the final solution of a configuration is the last, and best, solution it finds within the 30-minute timeout. (Note that the quality of a solution is always determined by its cost, irrespective of whether the heuristic used to calculate it is cost-sensitive or not.) When discussing the three possible approaches to costs (cost-unaware search, purely cost-sensitive search, or LAMA's combination of distances and costs) we write **X**, $\mathbf{X_c}$, and $\mathbf{X_c^+}$ to denote the three cost approaches independently of the heuristics used.

We measure performance using the criterion employed at IPC 2008 (see Section 2.4). Each planner configuration is run for 30 minutes per task. After the timeout, a planner aggregates the ratio c^*/c to its total score if c is the cost of the plan it has found, and c^* is the cost of the best known solution (e. g., a reference solution calculated by the competition organisers, or the best solution found by any of the participating planners). Experiments were run on the hardware used in the competition, a cluster of machines with Intel Xeon CPUs of 2.66 GHz clock speed. The memory limit was set to 2 GB as in the competition.

8.3.1 Overview of Results

As discussed in Chapter 5, the purely cost-based FF/add configuration $\mathbf{F_c}$ solves significantly fewer tasks than its cost-unaware counterpart **F**. While $\mathbf{F_c}$ finds *higher-quality* solutions, this does not make up for its low coverage when measuring performance with the IPC criterion. Using landmarks in the cost-unaware search improves quality slightly, so that **FL** achieves the highest IPC

performance score amongst our configurations. When using the *cost-sensitive* FF/add heuristic, adding landmarks increases coverage substantially, while incurring only a small loss in quality. Iterated search improves the scores of all configurations significantly. Lastly, using the combination of cost and distance estimates in the heuristics ($\mathbf{X}_\mathbf{c}^+$) is superior to pure cost-based search when using iterated search. Together, using landmarks and the combination of cost and distance estimates ($\mathbf{FL}_\mathbf{c}^+$) achieves nearly the same performance as the **FL** configuration.

In the following, we support these findings with experimental data. In Section 8.3.2 (*Performance in Terms of the IPC Score*), we show that the cost-sensitive FF/add heuristic by itself scores poorly in terms of the IPC criterion, but that landmarks and the combination of cost and distance estimates together make up for this bad performance. Furthermore, our results demonstrate the magnitude of the impact that iterated search has on the IPC scores. In Section 8.3.3 we give results on coverage. In Section 8.3.4 (*Quality*), we present data showing that the purely cost-sensitive FF/add heuristic finds higher-quality plans than the cost-unaware FF/add heuristic in the first search, but that with iterated search, this difference all but disappears. Furthermore, the approach of using cost and distance estimates scores higher than the purely cost-based search when using iterated search. LAMA's approach of using landmarks and the combination of cost and distance estimates ($\mathbf{FL}_\mathbf{c}^+$) thus effectively mitigates the bad performance of the cost-sensitive FF/add heuristic.

8.3.2 Performance in Terms of the IPC Score

The scores of all planners scoring more than 100 points at IPC 2008 are shown in the top part of Table 8.1. Apart from LAMA, this includes a base planner run by the competition organisers (FF with a preprocessing step that discards action costs), the FF(h_a) and FF(h_a^s) planners by Keyder and Geffner (2008) and the C3 planner by Lipovetzky and Geffner (2009). The plans found by these planners have been obtained from the competition website (Helmert et al., 2008). As in Chapter 5 we have re-calculated the scores for the IPC planners to reflect new best solutions found in our experiments, showing the original total scores in parentheses in the last table row.

While the configuration $\mathbf{FL}_\mathbf{c}^+$ results in essentially the same planner as (the IPC version of) LAMA, we report its results again, as some minor corrections have been implemented in LAMA since the competition. In addition, the planner makes arbitrary decisions at some points during execution due to underlying programming library methods, leading to varying results. However, as Table 8.1 shows these differences between $\mathbf{FL}_\mathbf{c}^+$ and LAMA are very small. We have furthermore added columns to the table showing the hypothetical results for LAMA that would be obtained if its search (excluding the time spent on translation and preprocessing of the task) were slowed down by the constant factors 10 and 100, respectively. The use of these slowing factors is equivalent to stopping LAMA's search module after 3 minutes, or 18 seconds, respectively. The numbers show that LAMA still outperforms the other IPC planners even with a severe handicap, demonstrating that the good performance of LAMA is not mainly due to an efficient implementation.

The bottom part of Table 8.1 contains the results for our six experimental configurations after

Domain	IPC Planners					Slowed LAMA		
	Base	**C3**	**FF**(h_a)	**FF**(h^s_a)	**LAMA**	**×10**	**×100**	**FL^+_c**
Cyber Security	4	9	20	20	28	27	26	28
Elevators	21	16	9	10	20	20	17	22
Openstacks	21	10	8	8	27	27	26	27
PARC Printer	27	18	16	23	21	19	12	22
Peg Solitaire	20	20	21	23	29	29	26	29
Scanalyzer	24	23	24	24	26	25	22	26
Sokoban	21	18	15	18	24	22	15	23
Transport	18	6	15	14	27	25	21	26
Woodworking	14	24	22	22	25	24	17	24
Total	169	143	150	162	**227**	218	183	**227**
(Total IPC 2008)	(176)	(151)	(157)	(169)	(236)	(—)	(—)	(—)

Domain	First solutions						Final solutions (iterated search)					
	F	**F$_c$**	**F^+_c**	**FL**	**FL$_c$**	**FL^+_c**	**F**	**F$_c$**	**F^+_c**	**FL**	**FL$_c$**	**FL^+_c**
Cyber Security	20	24	24	20	28	27	23	24	25	26	28	28
Elevators	22	9	9	23	14	16	29	10	15	27	16	22
Openstacks	20	23	20	13	13	14	27	29	28	27	28	27
PARC Printer	20	16	16	23	21	21	23	16	16	24	22	22
Peg Solitaire	20	23	20	20	22	21	29	29	29	29	29	29
Scanalyzer	19	21	20	22	21	21	24	24	25	29	24	26
Sokoban	18	20	19	18	19	19	24	24	24	22	23	23
Transport	18	15	15	24	24	23	19	17	17	24	26	26
Woodworking	22	20	20	20	20	20	23	21	22	20	23	24
Total	180	171	162	182	**183**	182	220	194	201	**229**	217	227

Table 8.1: IPC scores (rounded to whole numbers) for planners scoring ≥ 100 points at IPC 2008 (top) and our 6 experimental configurations (bottom). Scores for IPC planners were re-calculated (see text). LAMA ×10 and ×100 refer to the results achieved by LAMA when slowed down by factors of 10 and 100, respectively. **FL^+_c** is essentially the same as the IPC planner LAMA.

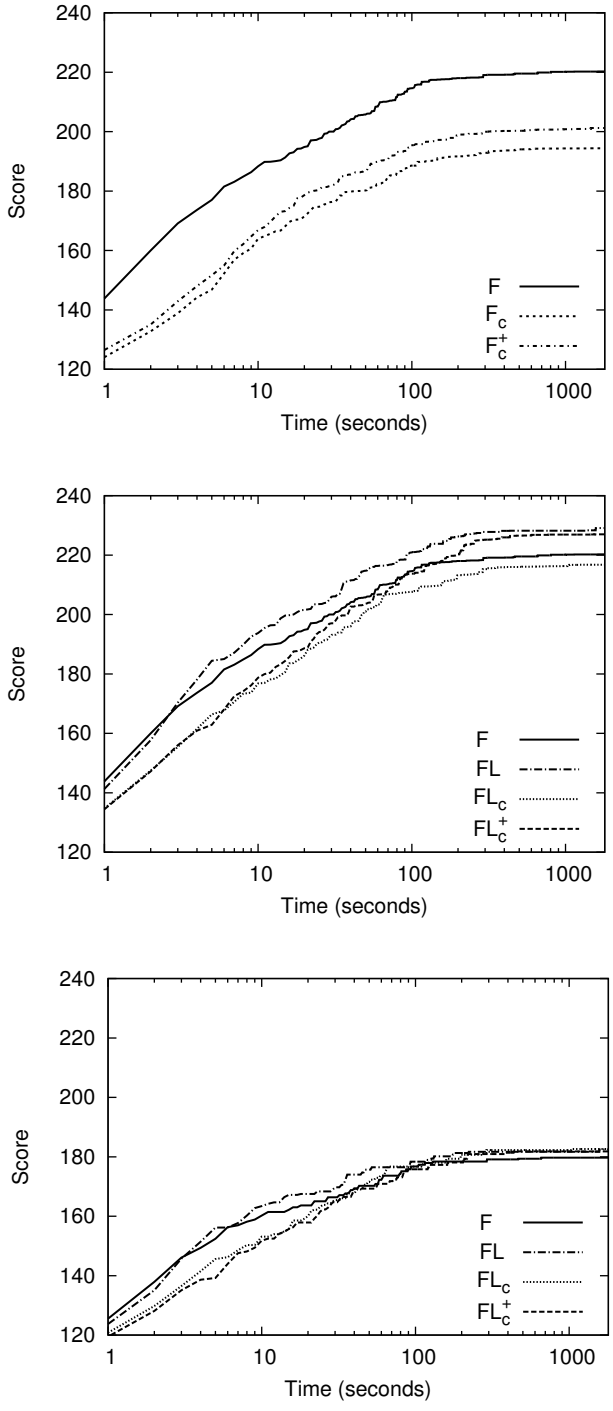


Figure 8.1: Score over time using iterated search (top and centre panel) and without iterated search, i. e., showing first solutions only (bottom panel).

the first search iteration (left) and after the 30-minute timeout (right). As can be seen, both the use of landmarks and iterated search lead to significant improvements in performance. Even with just one of those two features our planner performs notably better than any of its competitors from IPC 2008. In combination, the benefits of landmarks and iterated search grow further: in cost-unaware search the use of landmarks results in 2 additional score points for the first solutions, but in 9 additional points for the final solutions. Similar results hold for the cost-sensitive configurations. This is mainly due to the Openstacks domain, where using landmarks is highly detrimental to solution quality for the first solutions (see Chapter 5). Iterated search mitigates the problem by improving quality to similar levels with and without landmarks. Overall, there is thus a slight synergy effect between landmarks and iterated search, making the joint benefit of the two features larger than the sum of their individual contributions. The effect of landmarks in the Openstacks domain is discussed in more detail later.

Like in Chapter 5, the use of cost-sensitive search did not pay off in our experiments. Cost-unaware search is always at least roughly equal, and often substantially better than the cost-sensitive configurations, including the configurations that combine cost and distance estimates. Cost-based planning seems to be not only a problem for LAMA, but also for the other participating planners at IPC 2008: notably, all cost-sensitive competitors of LAMA fare worse than the cost-unaware baseline. In LAMA, best performance is achieved by using cost-unaware search with landmarks and iterated search. However, using the combination of cost and distance estimates instead ($\mathbf{FL}_\mathbf{c}^+$) leads to performance that is almost equally good. In particular, $\mathbf{FL}_\mathbf{c}^+$ is substantially better than the pure cost search $\mathbf{FL}_\mathbf{c}$ if iterated search is used.

A more detailed view on the same data is provided in Figure 8.1, where we show the performance over time for our six experimental configurations. A data point at 100 seconds, for example, shows the score the configuration would have achieved if the timeout had been 100 seconds. As the top panel shows, cost-sensitive search is consistently worse than cost-unaware search when using only the FF/add heuristic. Using landmarks (see centre panel), the two settings **FL** and $\mathbf{FL}_\mathbf{c}^+$ achieve better performance than **F**, though $\mathbf{FL}_\mathbf{c}^+$ needs 2 minutes to surpass **F**, while **FL** does so within 5 seconds. Pure cost search, even with landmarks ($\mathbf{FL}_\mathbf{c}$), performs worse than **F** at all times. The bottom panel of Figure 8.1 shows that when not using iterated search, the performance of the 4 best configurations **FL**, **F**, $\mathbf{FL}_\mathbf{c}^+$, and $\mathbf{FL}_\mathbf{c}$ is fairly similar eventually, but the cost-sensitive approaches need more time than the cost-unaware configurations to reach the same performance levels.

8.3.3 Coverage

As discussed in Chapter 5, the low IPC scores of the cost-sensitive FF/add heuristic is mainly due to low coverage. Table 8.2 shows the coverage for all considered planners and configurations. For $\mathbf{F}_\mathbf{c}$ and $\mathbf{F}_\mathbf{c}^+$ alike, using landmarks significantly improves coverage.

Domain	**Base**	**C3**	**FF**(h_a)	**FF**(h^s_a)	**LAMA**	**FL$_c^+$**
Cyber Security	4	15	23	22	30	30
Elevators	30	30	23	26	24	25
Openstacks	30	30	25	26	30	30
PARC Printer	30	18	16	23	22	23
Peg Solitaire	30	30	29	29	30	30
Scanalyzer	30	27	28	28	30	30
Sokoban	27	22	17	20	25	24
Transport	29	12	23	22	30	30
Woodworking	17	28	29	29	30	30
Total	227	212	213	225	251	**252**

Domain	**F**	**F$_c$**	**F$_c^+$**	**FL**	**FL$_c$**	**FL$_c^+$**
Cyber Security	30	28	29	30	30	30
Elevators	30	15	16	30	22	25
Openstacks	30	30	30	30	30	30
PARC Printer	25	16	16	24	23	23
Peg Solitaire	30	30	30	30	30	30
Scanalyzer	28	30	29	30	30	30
Sokoban	25	25	24	23	24	24
Transport	26	22	21	29	30	30
Woodworking	30	28	28	28	30	30
Total	254	224	223	**254**	249	252

Table 8.2: Coverage for planners scoring ≥ 100 points at IPC 2008 (top) and our 6 experimental configurations (bottom). Results of the IPC planners have been taken from the competition. **FL$_c^+$** is essentially the same as the IPC planner LAMA.

Domain	F_c / F		F_c^+ / F		FL_c / F_c		FL_c^+ / F_c^+	
	Tasks	**C. Ratio**	**Tasks**	**C. Ratio**	**Tasks**	**C. Ratio**	**Tasks**	**C. Ratio**
Cyber Security	28	0.64	29	0.69	28	0.81	29	0.83
Elevators	15	1.16	16	1.15	14	0.89	16	0.92
Openstacks	30	0.83	30	1.00	30	1.98	30	1.46
PARC Printer	16	0.79	16	0.79	15	1.05	15	1.05
Peg Solitaire	30	0.87	30	1.02	30	1.04	30	0.95
Scanalyzer	28	0.94	28	0.93	30	0.98	29	1.00
Sokoban	23	0.94	22	0.98	24	0.98	23	1.00
Transport	21	1.01	21	1.00	22	0.89	21	0.89
Woodworking	28	1.02	28	1.02	28	1.07	28	1.06
Average	219	0.88	220	0.94	221	1.06	221	1.01

Domain	F_c / F		F_c^+ / F		FL_c / F_c		FL_c^+ / F_c^+	
	Tasks	**C. Ratio**	**Tasks**	**C. Ratio**	**Tasks**	**C. Ratio**	**Tasks**	**C. Ratio**
Cyber Security	28	0.81	29	0.82	28	0.81	29	0.82
Elevators	15	1.51	16	1.05	14	0.89	16	0.99
Openstacks	30	0.95	30	0.96	30	1.04	30	1.04
PARC Printer	16	0.97	16	0.97	15	1.01	15	1.01
Peg Solitaire	30	0.99	30	1.00	30	1.02	30	1.01
Scanalyzer	28	1.01	28	0.93	30	1.02	29	1.01
Sokoban	23	1.01	22	1.00	24	1.02	23	1.00
Transport	21	0.98	21	0.89	22	0.89	21	0.89
Woodworking	28	0.99	28	0.94	28	1.00	28	1.01
Average	219	0.99	220	0.94	221	0.97	221	0.97

Table 8.3: Average ratio of the first solution costs (top) and best solution costs after iterative search (bottom) for various pairs of configurations on their commonly solved tasks. Geometric means were used for averaging.

8.3.4 Quality

In Chapter 5 we saw that the improvement in coverage achieved by landmarks in the cost-sensitive search comes at a price in solution quality, and that $\mathbf{FL_c}$ provides a middle ground between $\mathbf{F_c}$ and $\mathbf{F}$ when considering both coverage and solution quality. In this section, we look at how iterated search and the method of combining distance and cost estimates (the $\mathbf{X_c^+}$ configurations) influence quality. In particular, how much quality do we lose by combining distance and cost estimates ($\mathbf{X_c^+}$) as opposed to using pure cost search ($\mathbf{X_c}$)? The IPC score incorporates both coverage and quality information by counting unsolved tasks as 0 – a method that allows ranking several planners solving different subsets of the total benchmark set. When we are interested in examining quality independent of coverage, we must restrict our focus to those tasks solved by all compared planners. Table 8.3 contains quality information comparing the solution costs of several configurations, where we compare configurations pair-wise in order to maximise the number of commonly solved tasks. The top part of Table 8.3 contains comparisons involving the *first* solutions found by each configuration, while the bottom part of the table concerns the *best* solutions found after iterative search. For each pair of configurations we show the number of tasks solved by both, and the geometric mean of the *cost ratios* for the plans they find.

Landmarks deteriorate quality for the first plans of $\mathbf{F_c}$; but $\mathbf{F_c^+}$, which starts out with a worse quality than $\mathbf{F_c}$, is not noticeably further deteriorated by landmarks. For both configurations, however, the main negative impact through landmarks is in the Openstacks domain. By contrast, in the remaining 8 domains average plan quality for both configurations with landmarks is even slightly *better* on average than without landmarks.

We note that iterative search has a remarkable impact on the relative performance of the different configurations. When looking at the solutions found after iterative search, $\mathbf{F_c}$ actually performs worse than $\mathbf{F_c^+}$, whereas it is the other way round for the first solutions (compare the first two columns in the top row versus the bottom row of the table). This can be explained to some extent by the fact that the same reasons that cause $\mathbf{F_c}$ to have low coverage also prevent it from improving much over time. As we have showcased for the Openstacks domain in Chapter 5, cost-sensitive heuristic often expands many more nodes than the cost-unaware search, leading to the observed behaviour. This is most likely due to the fact that finding plans of high quality is hard and thus unsuccessful in many of the benchmark tasks.

With iterative search, landmarks do not deteriorate quality for either $\mathbf{F_c}$ nor $\mathbf{F_c^+}$ on average, as the negative impact of the Openstacks domain is no longer present. (This effect in the Openstacks domain will be discussed in more detail later.)

Summarising our findings, we can say that landmarks effectively support the cost-sensitive FF/add heuristic in finding solutions, without steering the search away from good solutions. Similarly, combining distance and cost estimates as in $\mathbf{X_c^+}$ leads the search to finding solutions quickly without overly sacrificing quality, as is demonstrated by its superior anytime performance compared to pure cost search.

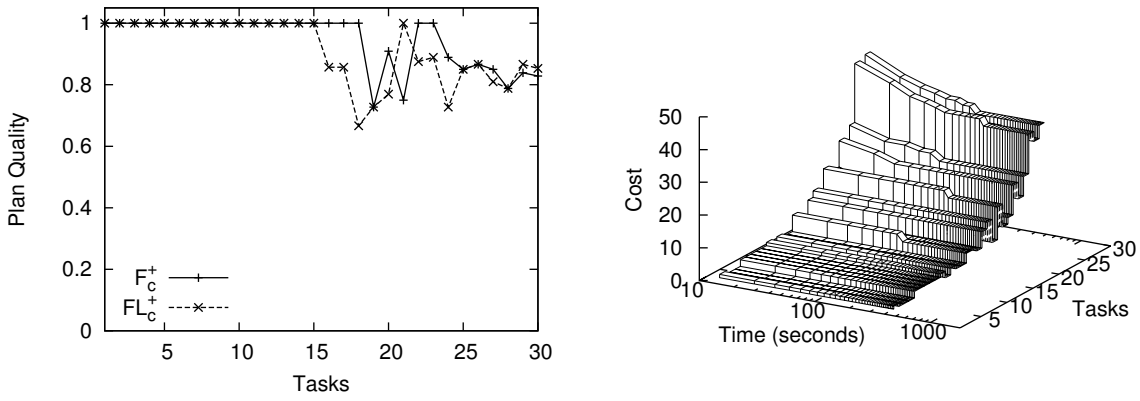


Figure 8.2: Effect of iterative search in the Openstacks domain. Left: plan quality (IPC score) of the best plan found within 30 minutes with and without landmarks. Right: evolution of plan costs with landmarks ($\mathbf{FL_c^+}$) over time.

8.3.5 Openstacks: Synergy between Landmarks and Iterated Search

As shown in Table 8.3, iterated search mitigates the bad effect of landmarks in the Openstacks domain: without iterated search, landmarks greatly deteriorate plan quality in Openstacks, however, with iterated search this does not hold. Detailed results showing this effect are given in Figure 8.2. As discussed in Chapter 5, using landmarks significantly decreases the number of expanded search nodes for pure cost search (the $\mathbf{X_c}$ configurations) in Openstacks, and the same is true for our $\mathbf{X_c^+}$ configurations. Using landmarks thus speeds up planning, allowing iterative search to effectively improve solution quality in the given time limit such that the final results of using landmarks are similar to those of not using landmarks.

8.3.6 Domains from Previous Competitions

In Tables 8.4 and 8.5 we compare LAMA against *cost-unaware* state-of-the-art planners, using the IPC domains from 1998 to 2006 except for the trivial Movie domain. Consequently, the cost-sensitive configurations of LAMA are not applicable. The configurations we examine for LAMA here are thus **FL** and **F**, both with iterated search and without, where **FL** with iterated search is shown as LAMA. We compare against FF and Fast Downward, for which we use current versions. In particular Fast Downward has evolved substantially since its 2004 competition version, the original causal graph heuristic having been replaced with the better context-enhanced additive heuristic (Helmert and Geffner, 2008). After correspondence with the authors, the version of Fast Downward used here is the one featuring in recent work by Richter and Helmert (2009).

As Table 8.4 shows, LAMA performs better than both FF and Fast Downward in terms of the IPC 2008 criterion. This is true even if we turn off landmarks or iterated search in LAMA, but not if we turn off both options simultaneously. Note that on these domains, no external reference plans were used for the calculation of the score, which may skew results slightly in favour of the configuration with higher coverage, and among configurations of equal coverage, in favour of the

one with better plan quality (see Chapter 6).

Table 8.5 shows that LAMA's edge over Fast Downward is due to higher-quality solutions rather than coverage, as Fast Downward solves more tasks. Compared to FF, LAMA has better coverage, with the gap between LAMA and FF being substantially larger than the gap between LAMA and Fast Downward. Note that the **F** and "LAMA" configurations roughly correspond to the planners shown as "base" and "heur" in Chapter 4. However, there are differences between the implementations, and changes that we made in order to support action costs affect performance in the Philosophers domain, where we observe a significant decrease in coverage. This is also one of the reasons for the difference in coverage between LAMA and the closely related Fast Downward system.

Comparing the various experimental configurations for LAMA, we note that the use of landmarks leads to moderate improvements in both coverage and solution quality. As mentioned above, iterative search significantly improves performance in terms of the IPC 2008 score.

8.4 Summary

We have given a detailed account of the LAMA planning system. The system uses two heuristic functions in a multi-heuristic state-space search: a cost-sensitive version of the FF heuristic, and a landmark heuristic guiding the search towards states where many subgoals have already been achieved. Action costs are employed by the heuristic functions to guide the search to cheap goals rather than close goals, and iterative search improves solution quality while time remains.

We have conducted an extensive experimental study on the set of benchmark tasks from the International Planning Competition 2008, in order to identify how much each of the features of our planner contributes to its performance in the setting of planning with action costs.

While the cost-sensitive FF/add heuristic leads to poor coverage, landmarks and the combination of cost and distance estimates mitigate this problem. Nevertheless, LAMA would still have achieved a slightly higher score at IPC 2008 if it had simply ignored costs, rather than using cost-sensitive heuristics. For *cost-unaware* search, we found landmarks to improve coverage and solution quality in the domains from the IPCs 1998–2006. On the domains from IPC 2008, landmarks improve solution quality for the cost-unaware search, but do not further increase the (already very high) coverage.

Iterative search improves results notably for all of our experimental configurations, raising the score of LAMA by a quarter on the IPC 2008 domains. In the Openstacks domain, we could furthermore observe a synergy effect between the iterative search and landmarks. While landmarks usually improve quality, in this domain they lead to bad plans by not accounting for action costs. However, they speed up planning so that the planner evaluates substantially fewer states. Iterative search then effectively improves on the initial bad plans while benefiting from the speed-up provided by the landmarks. In general, we can use landmarks as a means to quickly find good solutions, while using iterative search as a way to improve plan quality over time. Overall, we

Domain	FF	F. Downw.	LAMA	F	FL^{first}	F^{first}
Airport (50)	35	39	35	33	35	33
Assembly (30)	29	28	30	30	29	29
Blocks (35)	30	17	33	34	22	17
Depot (22)	20	13	16	15	13	12
Driverlog (20)	13	14	19	20	16	15
Freecell (80)	69	66	73	75	62	65
Grid (5)	4	4	5	5	4	4
Gripper (20)	20	15	20	20	20	18
Logistics 1998 (35)	35	33	34	33	33	32
Logistics 2000 (28)	28	25	28	28	28	28
Miconic (150)	150	118	150	143	150	117
Miconic Full ADL (150)	124	95	136	136	107	107
Miconic Simple ADL (150)	140	105	148	150	117	113
MPrime (35)	28	34	35	33	31	29
Mystery (30)	14	18	19	16	18	14
Openstacks (30)	29	29	29	30	29	30
Optical Telegraphs (48)	12	4	2	2	2	2
Pathways (30)	19	28	28	27	28	27
Philosophers (48)	11	48	29	34	29	34
Pipesworld Notank. (50)	25	31	43	42	26	27
Pipesworld Tank. (50)	16	28	36	38	27	28
PSR Small (50)	41	49	50	50	49	49
Rovers (40)	38	35	39	39	37	37
Satellite (36)	35	30	33	31	32	27
Schedule (150)	99	132	147	139	137	129
Storage (30)	16	16	19	20	16	18
TPP (30)	23	26	30	29	28	27
Trucks (30)	10	13	13	16	12	15
Zenotravel (20)	19	17	19	20	18	18
Total (1482)	1132	1113	**1300**	1288	1155	1099
Average (%)	68	69	**77**	**77**	70	67
PSR Large (50)	—	26	28	16	22	14
PSR Middle (50)	—	40	50	41	37	35

Table 8.4: IPC scores (rounded to whole numbers) in unit-cost planning for FF, Fast Downward and LAMA as well as experimental alternative configurations of LAMA (**F**: without landmarks, $\mathbf{FL}^{\text{first}}$: without iterated search, $\mathbf{F}^{\text{first}}$: without landmarks and without iterated search). Shown are total scores for each domain and a weighted average of the percentual scores (giving equal weight to each domain).

Domain	FF	F. Downw.	LAMA	F
Airport (50)	37	40	36	34
Assembly (30)	30	30	30	30
Blocks (35)	31	35	35	35
Depot (22)	22	19	17	16
Driverlog (20)	15	20	20	20
Freecell (80)	80	79	79	78
Grid (5)	5	5	5	5
Gripper (20)	20	20	20	20
Logistics 1998 (35)	35	35	35	35
Logistics 2000 (28)	28	28	28	28
Miconic (150)	150	150	150	150
Miconic Full ADL (150)	136	139	137	138
Miconic Simple ADL (150)	150	150	150	150
MPrime (35)	34	35	35	35
Mystery (30)	16	19	19	16
Openstacks (30)	30	30	30	30
Optical Telegraphs (48)	13	5	2	2
Pathways (30)	20	29	29	28
Philosophers (48)	13	48	29	34
Pipesworld Notank. (50)	36	43	44	43
Pipesworld Tank. (50)	21	38	38	40
PSR Small (50)	41	50	50	50
Rovers (40)	40	39	40	40
Satellite (36)	36	35	34	31
Schedule (150)	133	150	150	144
Storage (30)	18	18	19	20
TPP (30)	28	30	30	30
Trucks (30)	11	15	13	16
Zenotravel (20)	20	20	20	20
Total (1482)	1249	**1354**	1324	1318
Average (%)	74	**81**	79	79
PSR Large (50)	—	31	29	16
PSR Middle (50)	—	50	50	41

Table 8.5: Coverage in unit-cost planning for FF, Fast Downward and LAMA as well as the experimental **F** configuration of LAMA without landmarks. Shown is the total number of tasks solved for each domain and a weighted average of the percentual coverage (giving equal weight to each domain).

found that the domains used at IPC 2008 constitute a varied benchmark set that reveals various strengths and weaknesses in our planning system.

9

Sound and Complete Landmarks for And/Or Graphs

Since verifying that a fact is a landmark is PSPACE-complete, existing practical procedures for computing landmarks have targeted relaxed versions of the task at hand, specifically the delete relaxation Π^+ (see Chapter 3). Building on a novel characterisation of the LM^{ZG} algorithm, we here propose a method for computing the landmarks of general *and/or graphs*. Applied to planning, this approach allows us to find new forms of higher-order landmarks (namely *conjunctions* of facts) and landmarks that take into account delete information. We demonstrate that our approach finds strictly more *causal* landmarks than previous methods, but is costly to compute. We then discuss the relationship between increased computational effort and experimental performance, using these landmarks in a landmark heuristic similar to the one proposed in Chapter 4.

9.1 Introduction

In Chapter 4, we presented a method that uses landmarks to derive a heuristic function for a classical planning task, where the heuristic value is given by the number of landmarks that remain to be achieved. Since its first introduction (Richter et al., 2008), various extensions of this idea have been proposed in the recent literature. For example, an adaption to optimal planning (Karpas and Domshlak, 2009) has led to a state-of-the-art admissible heuristic. This admissible heuristic uses a cost-partitioning approach, where each landmark is associated with a cost that results from distributing the action costs of its achievers among all the landmarks these achievers can possibly make true. The heuristic is complemented with an improved version of the optimal search algorithm A*, LM-A*, which checks whether the landmarks achieved along some path to a given state are necessarily achieved along *all* paths to that state. This information is used to boost the set of landmarks that remain to be achieved (and thus the heuristic estimate for the state) while maintaining admissibility. For a more extensive description of this approach see Section 10.2.

In this chapter, we investigate one possible way of extending our previous work by focusing on more complex types of landmarks (i. e., landmarks that are not facts or disjunctions), and landmarks that are not necessarily delete-relaxation landmarks. We make use of the LM^{ZG} algorithm (see Section 3.1.4) that allows identifying the *complete* set of *causal* delete-relaxation landmarks in polynomial time. We give a declarative characterisation of this set of landmarks and show that

LM^{ZG} computes the landmarks described by our characterisation.

Building on this, we observe that the LM^{ZG} procedure can be applied to any (STRIPS) task that has no delete effects. Specifically, we take advantage of the recent Π^m compilation for planning tasks (Haslum, 2009). This compilation results in a task with no delete effects that encodes information about the delete effects of the original task. We describe this method in some more detail below. This allows us to obtain, for the first time, both *conjunctive landmarks* and *landmarks beyond the delete relaxation*. The runtime of our method is polynomial in the size of the compiled task, which grows exponentially in m. Furthermore, for sufficiently large m the landmarks computed are the complete set of causal landmarks for Π.

However, preliminary experimental results indicated that our method is too costly to be of use within a landmark heuristic for satisficing planning. We thus evaluate our landmark-detection approach experimentally in *optimal planning*, using the admissible landmark heuristic by Karpas and Domshlak discussed above. In heuristic-search approaches for optimal planning, optimality of a given plan is proven by exhausting all nodes in the search space that could lead to a cheaper plan. In this setting, heuristic accuracy is more important than in satisficing planning, and larger preprocessing times as well as per-node computation times may pay off if they lead to a significant reduction of the search space.

The notion of the delete relaxation plays a central role in this chapter. In the following, we thus use the terminology of the STRIPS formalism, in which the delete relaxation can be more conveniently expressed than in the general finite-domain representation (see Chapter 2).

9.2 Landmarks for And/Or Graphs

In order to give a more general declarative characterisation of the landmarks computed by the LM^{ZG} procedure of Zhu and Givan (2003), we first discuss AND/OR graphs and how the delete relaxation can be understood as an instance of this type of graph. A more complete treatment of the subject can be found in an article by Mirkis and Domshlak (2007).

An AND/OR graph $\mathcal{G} = \langle V_{\text{I}}, V_{\text{and}}, V_{\text{or}}, E\rangle$ is a directed graph with vertices $V := V_{\text{I}} \cup V_{\text{and}} \cup V_{\text{or}}$ and edges E, where V_{I}, V_{and} and V_{or} are disjoint sets called *initial nodes*, *AND nodes* and *OR nodes*, respectively. A subgraph $J = \langle V^J, E^J\rangle$ of $\mathcal{G}$ is said to *justify* a set of "goal nodes" $V_{\text{G}} \subseteq V$ if and only if the following are true of J:

1. $V_{\text{G}} \subseteq V^J$
2. $\forall a \in V^J \cap V_{\text{and}} : \forall \langle v, a\rangle \in E : v \in V^J \wedge \langle v, a\rangle \in E^J$
3. $\forall o \in V^J \cap V_{\text{or}} : \exists \langle v, o\rangle \in E : v \in V^J \wedge \langle v, o\rangle \in E^J$
4. J is acyclic.

Intuitively, J is a justification for V_{G} if J contains a "proof" that all nodes in V_{G} are "true" under the assumption that all nodes in V_{I} are true. The set V^J represents the nodes that are proven

to be true by J, and the edges E^J represent the arguments for why they are true. The four conditions then state that (1) all nodes in V_G must be proven true, (2) AND nodes are proven true if all their predecessors are true, (3) OR nodes are proven true if they have some true predecessor, and (4) the proof must be well-founded.

The delete relaxation can be understood as specifying an AND/OR graph in which the facts in the initial state constitute the initial nodes, other facts constitute OR nodes, and actions constitute AND nodes (Mirkis and Domshlak, 2007). Edges then correspond to the relations between the facts and actions described by the preconditions and add effects of each action, with a directed edge from an AND node a to an OR node f when $f \in add(a)$, and from f to a when $f \in pre(a)$. Relaxed plans are then justifications for the goal set. This graph differs from the relaxed planning graph in that it only contains a single copy of each fact and action. The relaxed planning graph corresponds to an unrolled version of this graph in which a copy of a node appears in every level of the graph after the first level in which it appears.

Many problems related to the delete relaxation can be understood as computations on this AND/OR planning graph. For example, the h^+ heuristic is the cost of the lowest-cost justification J for the goal $s_\star$, where the cost of J is defined as the sum of the costs of the actions corresponding to the AND nodes it contains. Similarly, the $h^{\max}$ heuristic (see Section 2.2.2) is the minimum, over all justifications J for $s_\star$, of the cost of the most costly path $\langle f_1, a_1, f_2, \ldots, a_{n-1}, f_n \rangle$ in J, where $f_1 \in V_I$, $f_i \in V_{or}$ for $i \neq 1$, $f_n \in s_\star$, and $a_i \in V_{and}$, and where the cost of a path is defined as above.

Definition 9.1. ***AND/OR landmarks***
Given an AND/OR graph $\mathcal{G} = \langle V_I, V_{and}, V_{or}, E \rangle$, a node n is a landmark for $V_G \subseteq V_I \cup V_{and} \cup V_{or}$ if $n \in V^J$ for all justifications J for V_G.

Intuitively, the landmarks for a set V_G in an AND/OR graph can be computed by considering the intersection of the vertex sets of all justifications for V_G. However, as the number of possible justifications is exponential, this method is intractable. The landmarks for V_G can also be characterised by the following system of equations:

$$
\begin{aligned}
LM(V_G) &= \bigcup_{v \in V_G} LM(v) \\
LM(v) &= \{v\} && \text{if } v \in V_I \\
LM(v) &= \{v\} \cup \bigcap_{u \in pred(v)} LM(u) && \text{if } v \in V_{or} \\
LM(v) &= \{v\} \cup \bigcup_{u \in pred(v)} LM(u) && \text{if } v \in V_{and}
\end{aligned}
$$

where $pred(v) = \{u \mid \langle u, v \rangle \in E\}$. The following theorem establishes the correctness of this characterisation.

Theorem 9.1. ***Correctness of the LM(·) characterisation***
For any AND/OR graph $\mathcal{G}$, the system of equations $LM(\cdot)$ has a unique maximal *solution, where maximal is defined with regard to set inclusion, and this solution satisfies*

$$u \in LM(v) \iff u \text{ is a landmark for } \{v\} \text{ in } \mathcal{G}.$$

Moreover, for any node set V_G, $LM(V_G)$ is the set of landmarks for V_G in $\mathcal{G}$.

Proof. Let $LM_c(v)$ denote the complete set of landmarks for v. A solution to the system of equations exists, as it is satisfied by setting $LM(v) = LM_c(v)$ for all v. To show that LM_c is the unique maximal solution, we show that all solutions to $LM(\cdot)$ satisfy $u \in LM(v) \Rightarrow u \in LM_c(v)$. Define a counterexample X as a tuple $\langle u, v, J\rangle$ such that J is a justification for $\{v\}$, $u \in LM(v)$, $u \notin J$. Assume a counterexample exists and choose one where $|X| := |V^J|$ is minimal. Whether $v \in V_{\text{and}}$ or $v \in V_{\text{or}}$, it is possible to choose a predecessor v' of v such that $u \in LM(v')$. Then $\langle u, v', J \setminus \{v\}\rangle$ is a counterexample X' with $|X'| < |X|$, contradicting the minimality of $|X|$. Hence, no counterexample exists. This shows that u must be contained in all justifications for $\{v\}$, which implies $u \in LM_c(v)$. ∎

Application to planning. The unique maximal solution to the $LM(\cdot)$ equations can be found in polynomial time with a generalised Dijkstra algorithm, analogously to the way the additive heuristic h^{add} may be calculated (see Chapter 2.2.2). One way to compute the solution is to perform a fixpoint computation in which the set of landmarks for each vertex is initialised to the set of all vertices of the graph $\mathcal{G}$ and then iteratively updated by interpreting the equations as update rules. If the updates are performed according to the order in which nodes are generated in the relaxed planning graph (i. e., all nodes in the first layer, then all nodes in the second layer, etc.), then we obtain exactly the label-propagation algorithm by Zhu and Givan (2003), computing action landmarks as well as causal fact landmarks. If only fact landmarks are sought, the equation for AND nodes can be modified to not include $\{v\}$ in $LM(v)$. The causal fact landmarks for a planning task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}\rangle$ computed via our equations then correspond to $LM(\Pi)$ defined as follows:

$$\begin{aligned}
LM(\Pi) &= LM(s_\star; s_0) \\
LM(F; s) &= \bigcup_{f \in F} LM(f; s) \\
LM(f; s) &= \begin{cases} \{f\} & \text{if } f \in s \\ \{f\} \cup \bigcap_{a \in A_f} LM(a; s) & \text{otherwise} \end{cases} \\
LM(a; s) &= LM(pre(a); s)
\end{aligned}$$

Orderings. Orderings for AND/OR landmarks can be defined analogously to orderings for planning landmarks, and they can be easily inferred from the LM sets. In particular, if u and v are two landmarks, we obtain a natural order $u \to v$ whenever $u \in LM(v)$.

For AND/OR graphs that represent delete relaxations, greedy-necessary orderings can also be computed with a slight extension. Let the set of *first achievers* for an OR node (fact) be defined as $FA(f) := \{a \mid a \in pred(f) \wedge f \notin LM(a)\}$. We can then infer $f \rightarrow_{\text{gn}} f'$ whenever $f \in pred(a)$ for all $a \in FA(f')$. Intuitively, this rule states that f is ordered greedy-necessarily before f' if f is a precondition for all actions that can possibly achieve f' for the first time. These orderings can be discovered during the computation of the landmarks.

9.3 Landmarks from the Π^m Task

One method for estimating the cost of the delete relaxation is the previously mentioned $h^{\max}$ heuristic, which recursively estimates the cost of a set of facts as the cost of the most expensive fact in the set (see Section 2.2.2). The $h^{\max}$ heuristic is a member of a more general formulation, the parameterised h^m family of heuristics which recursively estimate the cost of a set of facts F as the cost of the most expensive subset of F with size at most m (Haslum and Geffner, 2000). For $m = 1$, this coincides with the $h^{\max}$ heuristic. For $m > 1$, this heuristic takes into account delete information in the task, as a fact cannot be achieved in the context of a set to which it belongs with an action that deletes some other fact in the set.

It was recently shown that the h^m cost of a task Π can be computed as the h^1 cost of a task Π^m that results from a transformation of Π (Haslum, 2009). The facts of the new task Π^m represent all sets of facts of size m or less in the original task. Its actions are obtained by making explicit in the precondition and add effects of the original actions those facts which, while not required or added by an action, may occur in the state in which the action is applied and persist after the application of the action, allowing them to be achieved in conjunction with the effects of the action. This is done by creating for each action a in Π a *set* of actions in the new task, each having as a precondition in addition to the precondition of a itself, a set of facts C of size at most $m - 1$ such that C is disjoint from $add(a)$ and $del(a)$. For a set C and action a, the action a_C is then given by:

$$\begin{aligned} pre(a_C) &= \{S \mid S \subseteq (pre(a) \cup C) \wedge |S| \leq m\} \\ add(a_C) &= \{S \mid S \subseteq (add(a) \cup C) \wedge |S| \leq m\} \\ del(a_C) &= \emptyset \end{aligned}$$

Π^m is a task with no delete effects that nevertheless encodes in its facts and actions some of the information about delete effects specified in the original task. Any procedure applicable to a delete relaxation task Π^+ can also be applied to Π^m to obtain information that can be translated back into the facts and actions of the original task and used in that setting. In particular, the solution to the set of equations given above when the input is the Π^m task defines *conjunctive* landmarks of size m or less that take into account *delete information* in the original task Π.

Just as the h^m family of heuristics approaches optimality as m goes to infinity (Haslum and Geffner, 2000), it can be shown that the set of landmarks computed by the above procedure for

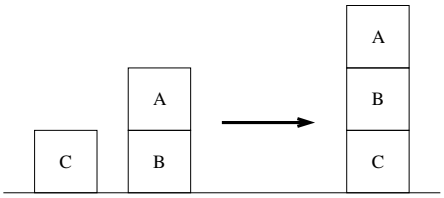


Figure 9.1: A Blocksworld task.

Π^m will approach the complete and sound set of causal landmarks for the original task Π. Yet since the complexity of computing Π^m and its size grow exponentially in m, this is unlikely to be feasible for high values of m.

Example. Consider the Blocksworld task of Figure 9.1. Apart from trivial landmarks such as those facts belonging to the initial state or goal, the complete set of causal delete-relaxation landmarks and orderings is $\mathit{clear}\ B \rightarrow_{\mathrm{gn}} \mathit{holding}\ B$, implying that *holding B* must be made true in some state by every plan, and that *clear B* must be true in the state that immediately precedes it. In contrast, when the landmarks computation is applied to the Π^2 compilation of the task, one of the obtained chains of orderings is the following:

$$(\mathit{clear}\ B \wedge \mathit{holding}\ A) \rightarrow_{\mathrm{gn}} (\mathit{clear}\ B \wedge \mathit{handempty}) \rightarrow_{\mathrm{gn}}$$
$$(\mathit{holding}\ B \wedge \mathit{ontable}\ A) \rightarrow_{\mathrm{gn}} (\mathit{on}\ B\ C \wedge \mathit{ontable}\ A) \rightarrow_{\mathrm{gn}}$$
$$(\mathit{on}\ B\ C \wedge \mathit{holding}\ A)$$

where $a \wedge b$ is a conjunctive landmark that implies that a and b must be true simultaneously in some state. These landmarks and orderings are only a subset of those found by the procedure, yet provide an almost complete roadmap for solving the task.

The additional landmarks found in this way are not only conjunctive: the consideration of delete effects may also result in the discovery of fact landmarks for Π that are not landmarks in the Π^+ task. In this example, the facts *holding A* and *ontable A* are also implied to be landmarks, as they are part of a conjunctive landmark.

9.4 Experimental Results

We implemented the Π^m transformation and the computation of landmarks as discussed in Section 9.2. Here, we try to answer three main questions: whether our approach finds landmarks not found by previous approaches, whether these landmarks contain interesting information, and finally, whether current planners can exploit this information. For our experiments, we use the STRIPS domains of IPCs 1998–2006 except for the trivial Movie domain, and the domains of the optimisation track of IPC 2008 (that are also expressed in STRIPS). All experiments were run on 2.3 GHz AMD Opteron machines using a 2 GB memory limit and 30-minute timeout. The results

Domain	# Causal LM Facts RHW	Avg. Ratio to RHW m = 1 (ZG)	m = 2 Facts	m = 2 Conj.
Airport (11)	1043	1.00	1.00	23.71
Blocks (35)	1444	1.00	1.05	8.36
Depot (21)	1379	1.07	1.13	13.07
Driverlog (20)	504	1.02	1.02	7.06
Freecell (46)	2788	1.28	1.30	14.61
Grid (4)	70	1.14	1.14	3.36
Gripper (20)	960	1.00	1.00	10.35
Logistics 1998 (23)	816	1.00	1.00	3.45
Logistics 2000 (28)	1319	1.00	1.00	4.02
Miconic (150)	7720	1.00	1.00	3.52
Mprime (25)	94	1.07	1.66	2.73
Mystery (16)	66	1.03	1.64	2.86
Openstacks (24)	2946	1.03	1.03	11.08
Pathways (30)	956	1.50	1.57	7.31
Pipesworld Notankage (44)	754	1.22	1.29	4.25
Pipesworld Tankage (26)	524	1.15	1.24	5.41
PSR Small (50)	550	1.00	1.60	7.32
Rovers (33)	733	1.17	1.19	6.62
Satellite (23)	515	1.01	1.01	7.31
TPP (24)	751	1.13	1.32	5.94
Trucks (14)	467	1.23	1.25	8.92
Zenotravel (19)	360	1.05	1.05	5.56
Elevators (30)	629	1.12	1.12	3.66
Openstacks (30)	2925	1.03	1.03	11.37
PARC Printer (30)	2142	1.00	1.07	18.37
Peg Solitaire (30)	1457	1.00	1.02	19.33
Scanalyzer (26)	673	1.00	1.26	9.65
Sokoban (29)	605	2.73	4.80	36.95
Transport (30)	390	1.00	1.00	3.44
Woodworking (30)	1520	1.06	1.08	9.91

Table 9.1: Total number of causal (fact) landmarks found by RHW in each domain, and average ratio between the causal landmarks found by our approach and by RHW. The geometric mean was used to average over the tasks from each domain. In the last column, conjunctive landmarks as well as facts are counted. Top part of table: STRIPS domains of IPCs 1998–2006. Only solvable tasks are listed for Mystery. Bottom part of table: domains of the optimisation track of IPC 2008. Numbers behind domain names show the number of tasks considered for that domain (the tasks where LM detection finished for all configurations).

Domain	RHW # Expansions	Improvement over RHW m = 1 (ZG)	m = 2 Facts	m = 2 Conj.
Airport (11)	385	1.00	1.00	1.12
Blocks (25)	4135432	1.00	1.00	9.80
Depot (4)	365373	1.07	1.48	3.75
Driverlog (8)	868496	1.00	1.00	1.02
Freecell (35)	178433	2.10	2.10	2.41
Grid (1)	270	1.50	1.50	1.64
Gripper (5)	458498	1.00	1.00	1.00
Logistics 1998 (3)	45663	1.00	1.00	1.48
Logistics 2000 (20)	862443	1.00	1.00	22.80
Miconic (141)	135213	1.00	1.00	1.34
Mprime (15)	313579	1.00	1.34	1.39
Mystery (12)	290133	1.00	1.00	1.00
Openstacks (7)	27386	1.00	1.00	1.00
Pathways (4)	152479	1.61	1.61	1.61
Pipesworld Notankage (16)	1931233	1.05	1.05	1.46
Pipesworld Tankage (8)	29698	1.00	1.00	0.91
PSR Small (48)	697969	1.00	1.03	1.62
Rovers (5)	231520	1.06	1.06	1.06
Satellite (6)	1292489	1.01	1.01	1.06
TPP (5)	12355	1.00	1.00	1.00
Trucks (2)	108131	1.02	1.02	1.05
Zenotravel (8)	186334	1.00	1.00	1.02
Elevators (7)	483982	1.00	1.00	1.35
Openstacks (10)	649341	1.00	1.00	1.00
PARC Printer (12)	1118898	1.00	1.29	1.61
Peg Solitaire (23)	1734655	1.00	1.04	1.20
Scanalyzer (11)	23029	1.00	1.00	1.46
Sokoban (13)	3502115	1.02	1.03	1.10
Transport (9)	929285	1.00	1.00	1.00
Woodworking (10)	199666	1.41	1.41	2.35

Table 9.2: Expanded states when using the landmark detection of RHW and average improvement ratios of our approach using the optimal cost partitioning method. Numbers behind domain names show the number of tasks considered for that domain (the tasks solved by all configurations).

reported here differ slightly from the ones we published in a recent conference article (Keyder et al., 2010), as we have since corrected an error in our implementation.

Number of Landmarks. Table 9.1 contrasts the number of causal landmarks we find with the causal fact landmarks found by the method we introduced in Chapter 3, denoted as RHW. (We removed all non-causal landmarks found by the RHW method in a post-processing step.) With $m = 1$, our Π^m approach is equivalent to the procedure by Zhu and Givan (2003), and in accordance with theory generates a superset of the causal fact landmarks that the RHW method finds, improving on the RHW method by 10–30% in several domains. With $m = 2$, we again generate a superset of the causal fact landmarks that $m = 1$ generates, improving on RHW by 10–60% in several domains. Particularly notable is the large number of conjunctive landmarks found with $m = 2$, surpassing the number of RHW facts by factors between 3 and 43. However, using $m = 2$ is computationally costly. Landmark detection with $m = 2$ timed out or ran out of memory in several cases in Airport and Freecell (as well as on large tasks in other domains that are far beyond the reach of current optimal planners).

Heuristic accuracy of landmark information. In order to assess how the additional landmarks may influence heuristic accuracy, we use them in the LM-A* algorithm using the admissible landmark counting heuristic of Karpas and Domshlak (2009), which we have extended to handle conjunctive landmarks. Cost partitioning among landmarks is performed optimally. Table 9.2 shows the number of expanded states in those tasks solved by all configurations. We show results both for the case in which $m = 2$ is used only to compute additional facts, and for when the additional conjunctive landmarks are used during planning. As can be seen, the number of expansions is improved in some domains by 30–50% even when using only the additional facts found with $m = 2$. With conjunctive landmarks, improvements of factors above 2 occur in several domains, with Logistics 2000 showing an improvement beyond factor 22.

Figure 9.2 compares the expansion data from Table 9.2 with the number of expansions resulting from *uniform* cost partitioning. While our approach expands significantly fewer nodes than RHW when used in combination with optimal cost partitioning, with uniform partitioning this advantage is smaller for $m = 2$ when using only facts, and all but disappears for $m = 2$ when also using conjunctive landmarks.

Planning performance. While optimal cost partitioning among landmarks leads to best heuristic accuracy, this method is unfortunately too costly to be competitive with the simpler uniform cost partitioning in terms of runtime and total number of tasks solved. In Table 9.3, we report the total number of tasks solved with each of our experimental configurations when using the uniform partitioning method. Domains where landmark detection with $m = 2$ was computationally too costly (timing out in tasks that were solved by RHW) are shown in parentheses at the bottom of the table and not included in the total. Our approach with $m = 1$ solves more tasks than RHW,

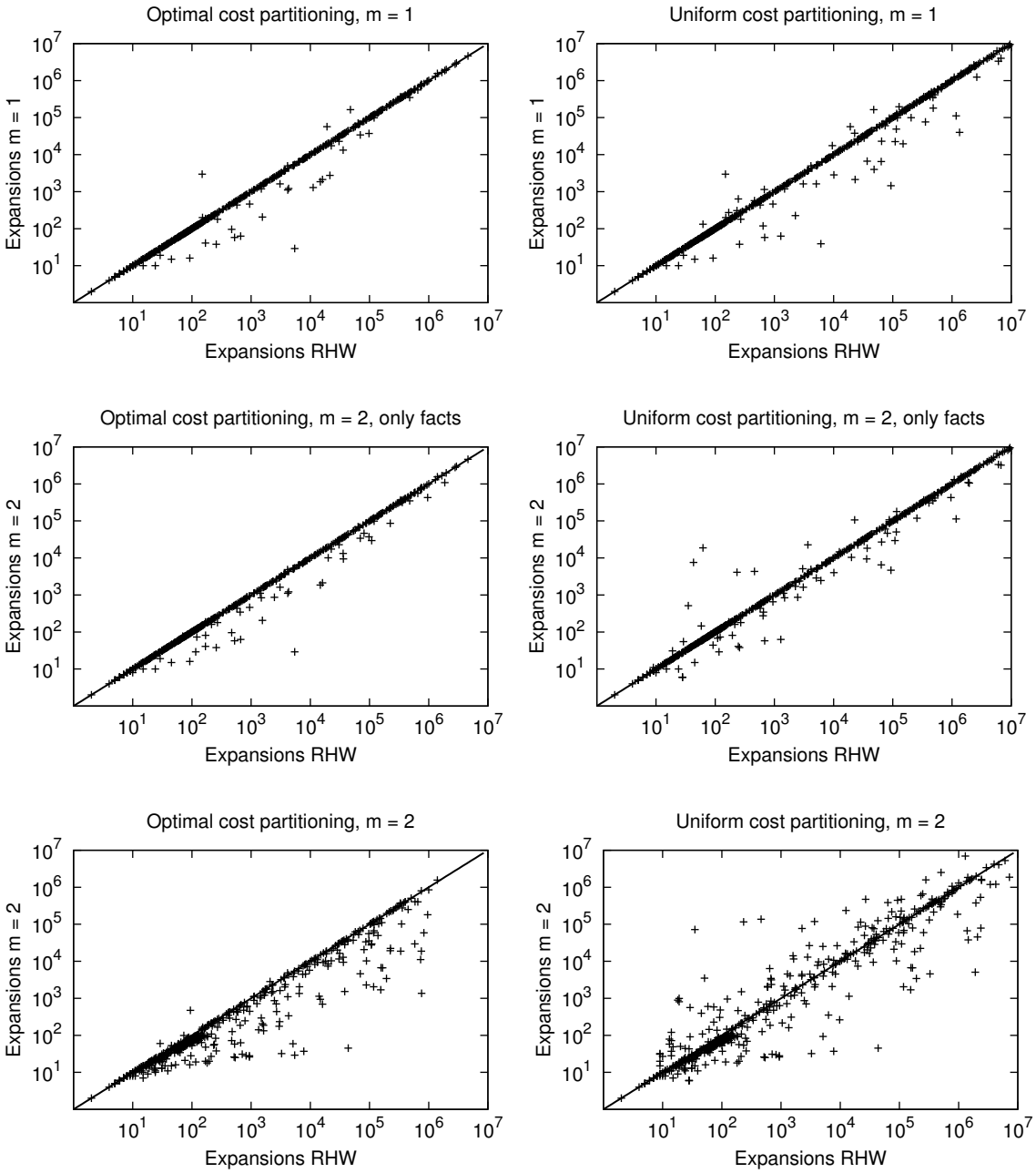


Figure 9.2: Expansions, compared to the RHW landmark detection (x-axes), of our approach using $m = 1$ (top), $m = 2$ when using only facts (middle), and $m = 2$ when using facts and conjunctive landmarks (bottom). Left column: optimal cost partitioning, right column: uniform cost partitioning.

Domain	RHW	m = 1 (ZG)	m = 2 Facts	m = 2 Conj.
Blocks (35)	26	26	26	28
Depot (22)	7	7	7	7
Driverlog (20)	10	10	10	9
Grid (5)	2	2	2	2
Gripper (20)	7	7	7	7
Logistics 1998 (35)	3	3	3	3
Logistics 2000 (28)	20	20	20	22
Miconic (150)	142	142	142	142
Mystery (19)	15	15	15	15
Openstacks (30)	7	7	7	7
Pathways (30)	4	4	4	4
Pipesworld Notankage (50)	19	19	19	18
Pipesworld Tankage (50)	12	13	13	11
PSR Small (50)	49	49	49	49
Rovers (40)	6	6	6	5
Satellite (36)	6	6	6	6
TPP (30)	6	6	6	6
Trucks (30)	2	2	2	2
Zenotravel (20)	8	8	8	8
Elevators (30)	13	13	13	14
Openstacks (30)	17	17	17	12
PARC Printer (30)	14	14	16	12
Peg Solitaire (30)	27	27	27	26
Scanalyzer (30)	9	9	9	6
Sokoban (30)	21	24	23	16
Transport (30)	11	11	11	11
Woodworking (30)	13	12	12	9
Total (951)	476	479	**480**	457
Average (%)	44	**45**	**45**	42
(Airport) (50)	(26)	(26)	(11)	(11)
(Freecell) (80)	(55)	(60)	(43)	(30)
(Mprime) (35)	(19)	(19)	(18)	(18)

Table 9.3: Solved tasks when using the landmark detection of RHW and our approach using the uniform cost partitioning method. Numbers behind domain names show the total number of solvable tasks in that domain. Also shown is a weighted average of the percentual coverage (weighting domains equally).

	RHW			m = 2 using conj. LMs		
Inst.	**LM**	**Exp.**	**Time**	**LM**	**Exp.**	**Time**
Logistics 2000						
5-0	33	936	0.03	33 + 66	28	0.00
7-0	44	7751	0.48	44 + 112	37	0.02
10-0	56	194038	18.58	56 + 192	3421	3.29
11-0	61	156585	19.58	61 + 221	6706	6.82
12-0	56	117387	14.63	56 + 236	2041	2.62
Depot						
2	29	1488	0.06	34 + 193	310	0.15
4	54	2347873	294.84	60 + 586	531785	877.38
7	42	167561	11.91	46 + 350	79627	63.03
10	47	1956533	174.40	55 + 456	375300	476.67
13	62	507369	64.64	62 + 623	336339	627.70
Driverlog						
3	10	1109	0.02	10 + 29	2105	0.09
5	17	247579	8.51	17 + 73	658799	63.90
7	17	26591	1.32	17 + 94	88915	16.59
10	14	504955	22.57	14 + 55	2506690	288.30
11	14	1298547	41.22	14 + 49	6969276	601.54

Table 9.4: Detailed results for selected domains, comparing $m = 2$ to RHW with respect to landmarks found, expanded states and runtime. Landmarks shown are causal facts for both approaches and conjunctive landmarks for $m = 2$ (second term in the sum).

and $m = 2$ using only facts solves one more task than $m = 1$. Using conjunctive landmarks during planning, however, does not pay off.

The coverage results in this table are not as good as one might expect when considering the improvement in expanded states shown in Table 9.2. The scatter plots in Figure 9.2 indicate that this can be largely attributed to the uniform cost partitioning method.

Table 9.4 shows detailed results for selected domains, demonstrating how the benefit of additional heuristic accuracy does not always pay off compared to the extra computational effort needed for generating and managing the conjunctive landmarks. While in Logistics 2000, our approach using $m = 2$ performs better than RHW both with respect to expansions and time, in Depot, $m = 2$ performs better with respect to expansions, but worse with respect to time. Driverlog is an example where the conjunctive landmarks are not helpful at all and RHW performs better both with respect to expansions and time. We also found that while having more causal fact landmarks *usually* translates to better heuristic accuracy, this is not always the case. Reasons for this when using the optimal cost-partitioning method include random effects due to the path-dependence of the heuristics (as different expansion orders may lead to different sets of accepted landmarks for the same state), and the fact that a dominating heuristic may perform worse in an A*algorithm due to tie-breaking in the last f-layer (Holte, 2010).

9.5 Conclusions and Future Work

We have shown how to declaratively define the complete set of causal landmarks for AND/OR graphs. Combined with the Π^m compilation, this results in a parameterised method that permits the computation of conjunctive and fact landmarks that take into account delete information in planning tasks. Our experimental results indicate that the use of these landmarks can significantly increase the accuracy of landmark-based admissible heuristics. However, computing these landmarks and using them is costly. The optimal cost-partitioning method is too complex to be competitive with the uniform cost-partitioning method in terms of coverage, and the uniform cost-partitioning scheme does not leverage the heuristic accuracy of the conjunctive landmarks sufficiently to be competitive with simpler landmark approaches.

There are thus two main avenues for future work in this area: finding higher-order landmarks more efficiently and using them more efficiently. As for the former, a major obstacle to finding higher-order landmarks in many tasks is the size of the Π^m compilation. It would thus be interesting to investigate complete and approximate methods for decreasing the size of the Π^m task by eliminating m-fluents that are irrelevant in the context of landmark detection. As for *using* higher-order landmarks more efficiently, our experimental evaluation showed a significant gap in heuristic quality between the optimal and uniform cost-partitioning schemes. A promising line of research is therefore to develop cost-partitioning schemes that offer favourable tradeoffs between the speed of the uniform scheme and the heuristic quality of the optimal scheme.

10

Conclusion

We have developed and evaluated several methods that now form part of the state of the art in automated planning. Here, we summarise our contributions and discuss the impact of our work by describing derivative work from the literature. Finally, we discuss possible directions of further research.

10.1 Summary

We have focused on improving automated planning using heuristic search in the state space, the most widely used approach to planning to date. We have developed several heuristics and algorithms for achieving good coverage and quality in this setting. The resulting LAMA planner is a state-of-the-art planning system that successfully trades runtime against quality. We have furthermore performed extensive evaluations that improve our understanding of several of the techniques used in LAMA, facilitating the use of these techniques in future research. Below we summarise our contributions.

The landmark heuristic proposed in Chapter 4, denoted by h^{LM} in the following, enables us to exploit automatically extracted control knowledge, specifically landmarks, in the search for a plan. In concert with existing planning heuristics, h^{LM} yields improvements both in terms of coverage and quality. Our method is an important step forward from the previous approach of Hoffmann et al. (2004). Indeed, the excellent results obtained using h^{LM} in the LAMA planner have rekindled interest in landmarks as a concept in developing effective planning procedures (see Section 10.2). We further developed a novel method of extracting landmarks via domain transition graphs and a generalisation of the backchaining algorithm to disjunctive landmarks. Consequently, our approach is able to compute more landmarks and orderings than the previous approaches by Hoffmann et al. (2004) and Zhu and Givan (2003).

Next, we examined how combinations of h^{LM} with variants of the FF/add heuristic affect the performance of planning in the cost-based setting. We found that the cost-sensitive FF/add heuristic, though known to lead to good plan quality (see Keyder and Geffner, 2008), is inferior to cost-unaware $h^{\mathrm{FF/add}}$ in our evaluation both in terms of coverage and according to the IPC 2008 performance criterion. We examined two domains in detail, determining characteristics of

planning tasks that can cause $h^{\text{FF/add}}$ to perform worse than its cost-unaware counterpart. We also demonstrated how using the landmark heuristic h^{LM} in addition to cost-sensitive $h^{\text{FF/add}}$ mitigates the coverage problems of the latter.

By conducting a detailed experimental comparison of various usages of preferred operators, we were able to demonstrate that the dual-queue method of using preferred operators gives best results among a number of principled alternatives in a greedy best-first search. Our results provide empirical evidence for claims in the literature that had been missing to date, and explain the good performance of Fast Downward at IPC 2004. Notably, we have shown that the right usage of preferred operators can be more important than the choice of heuristic.

Our anytime algorithm RWA* provides a way of achieving good plan quality given limited time. It first searches greedily with the aim of solving any given task (even hard tasks) within the allocated time. After the first solution, and every time a new best solution is found, the search becomes iteratively less greedy and more focused on solution quality. Our approach of restarting from the initial state between search iterations provides a significant improvement over existing anytime algorithms in our experiments, and we illustrate the reason for this effect that we call low-h bias.

In the LAMA planner, we combine several of the techniques we have previously investigated: the landmark heuristic with disjunctive landmarks, the search enhancements proven useful in our detailed study, and the restarting anytime search. In order to avoid an overly strong focus on plan quality at the cost of coverage, the LAMA planner combines cost and distance estimates within its heuristics. LAMA won the sequential satisficing track at IPC 2008, outperforming its competitors by a considerable margin. We analyse how LAMA's distinguishing features, namely the landmark heuristic, anytime search and the combination of cost and distance estimates, influence its performance on the competition benchmarks. This gives us insights into the reasons for LAMA's success and provides useful guidance for the design of future planning systems.

As an extension of our landmark approach, we conducted initial investigations into more complex forms of landmarks, namely landmarks that go beyond the delete relaxation. We found that these higher-order landmarks lead to greatly improved heuristic accuracy, but that the increased cost for finding and using these landmarks does not necessarily pay off. More work will be needed in order to mitigate the additional computational effort in handling these landmarks and derive a practical performance benefit.

10.2 Related Work

Following the publication of our work on the landmark heuristic and LAMA's victory at IPC 2008, various groups of researchers have become interested in the topic of landmarks.

Karpas and Domshlak (2009) adapt our landmark heuristic to optimal planning. Note that h^{LM} as introduced in Chapter 4 is not admissible, as an action may achieve several landmarks at once. Karpas and Domshlak make our landmark heuristic admissible, resulting in the heuristic h^{L}, via

cost partitioning (Katz and Domshlak, 2008a): first, the costs of actions that may achieve several landmarks are distributed among those landmarks. Then each landmark is assigned a cost that is no greater than the minimum cost assigned to it by any of its achievers. This cost partitioning is recomputed for each state expanded during the search, using all possible achievers or first achievers depending on whether a landmark has never been accepted or is required again. Karpas and Domshlak study both *uniform* cost partitioning, where the costs of actions are distributed evenly among the landmarks they achieve, and *optimal* cost partitioning, where the distribution leading to highest heuristic values is found via linear programming.

In the same work, the authors use action landmarks to improve their cost partitioning, resulting in the heuristic h^{LA}, as follows. The cost for any action landmark is accounted for, and all landmarks that might be achieved by these actions are removed from the landmark set before cost partitioning. This can only increase the final landmark costs and thus improve the heuristic. In addition, Karpas and Domshlak propose to make the heuristic *multi-path-dependent* and take into account *all* known paths to a state when calculating its heuristic value. A landmark that is not accepted along all such paths can safely be counted as not accepted in the state. A modified version of the A* algorithm called LM-A* is used to store the necessary information about all paths associated with the state, and recompute the heuristic value of a state each time a new path to the state is discovered. The resulting approach is shown to compete favourably with the state of the art in optimal planning.

Wang et al. (2009) view landmarks as *temporally extended goals*, express them in Linear Temporal Logic and encode them via finite-state automata. The authors then compile these automata via variables into the planning tasks, with the aim of allowing planning heuristics to reason about the landmarks and their orderings. Since the landmark discovery procedure they use can only find delete-relaxation landmarks, this approach cannot add any information to a delete-relaxation-based heuristic (the authors experiment with the FF heuristic). However, in a different setting the compilation approach has the potential of enriching heuristics with landmark information.

Cai et al. (2009) use landmark orderings as *precedence constraints* in an attempt to improve the context-enhanced additive heuristic. The authors report that their technique can lead to better results than h^{cea} depending on the choice of precedence constraints, but the improvement is moderate and none of the precedence constraints tested performed consistently better than h^{cea}.

Helmert and Domshlak (2009) establish theoretical results concerning the connections between delete-relaxation heuristics (such as $h^{\max}$,h^{add},h^{FF}) and landmark heuristics (h^{LM}, h^{L}, h^{LA}). They show that landmark heuristics can be compiled into *additive* $h^{\max}$ heuristics ($h^{\max}$ with suitable cost partitioning) in polynomial time, and vice versa (for states of finite $h^{\max}$ value). Building on these results, Helmert and Domshlak propose a new heuristic, the *landmark-cut heuristic* $h^{\text{LM-cut}}$, that results from compiling $h^{\max}$ into additive landmark heuristics. The landmark-cut heuristic compares extraordinarily well to current heuristics, advancing the state of the art in performance for optimal planning.

Bonet and Helmert (2010) analyse the landmark-cut heuristic introduced by Helmert and Dom-

shlak (2009) and establish theoretical results regarding the connection between the perfect delete-relaxation heuristic h^+ and landmarks. The authors show that h^+ is equivalent to the minimum-cost solution of a *hitting set problem* induced by the action landmarks of a task, and that $h^{\text{LM-cut}}$ is the optimal solution for a relaxation of this problem. Building on these results, the authors propose other ways of relaxing the hitting set problem to derive heuristics bounded by h^+ that dominate $h^{\text{LM-cut}}$. In preliminary experiments, however, these heuristics were significantly more costly to compute than $h^{\text{LM-cut}}$ and resulted in lower coverage despite their higher heuristic accuracy.

Domshlak et al. (2010) integrate landmark information into *abstraction heuristics* (Helmert et al., 2007; Katz and Domshlak, 2008b). They propose to compile the landmarks of a task into the task, thus making the implicit subgoals explicit goals in order to arrive at better-informed abstractions. The authors report substantially improved heuristic estimates, in particular with *structural-pattern heuristics* (Katz and Domshlak, 2008b), which are less sensitive to the increased task size that results from the compilation, compared to explicit-abstraction heuristics.

Helmert (2010) draws a connection between landmark heuristics and a gap-counting heuristic for the *pancake problem* (e. g., Dweighter, 1975; Gates and Papadimitriou, 1979), and shows that the gap heuristic outperforms other current approaches for the pancake problem.

Buffet and Hoffmann (2010) apply landmark heuristics to *probabilistic planning*. They find that heuristic estimates are substantially more accurate when landmark information is used, yet overall planning performance is not improved compared to a simple goal-counting heuristic. The authors conjecture that this is due to the heuristics not taking the probabilistic nature of the tasks into account (the tasks are determinised for heuristic computation), and that less informed heuristics lead to more random walks in the search space and, as a consequence, are less likely to make costly mistakes. Buffet and Hoffmann furthermore propose improvements to the landmark heuristic h^{LM}, for example by extending the notion of "required again" landmarks to encompass such landmarks that are currently true but will need to be made false and true again. This is the case for "doomed goals", i. e. landmarks that are goals and that will need to become false because they are inconsistent with landmarks that still need to be achieved.

Ridder and Long (2010) propose using landmarks in *partial-order planning* by splitting a given task into subtasks. While their preliminary experimental results are not competitive with the state of the art in planning, the authors suggest several possible improvements to their current approach.

In recent work on solving diagnosis problems via planning techniques (Anonymous, 2011, author names withheld due to blind reviewing process), the LAMA planner is compared to several other systems in an experimental study. The authors deem LAMA's performance to be the most satisfying among the compared planners. While cost-optimal methods fail to solve large tasks and cost-unaware approaches lead to poor solution quality, LAMA is noted for striking a favourable balance between coverage and solution quality. These results provide an interesting contrast to our findings concerning the IPC 2008 benchmarks, where LAMA's sensitivity to action costs did not result in any benefit over cost-unaware approaches.

Our approach of using restarts in a best-first search to overcome low-h bias has also resulted in follow-up work by other researchers. Vidal et al. (2010) adopt the idea of restarts for planning during parallel computation. Conducting a k-parallel best-first search on a multi-core machine, the authors increase the number of parallel threads iteratively while no solution is found, restarting the search each time the number of threads is increased. The authors report significant benefit of the restarting mechanism, resulting in a performance gain nearly as large as the difference between one thread and 64 threads.

Thayer and Ruml (2010) compare restarting anytime search with continued anytime search (without weight decreases) and repairing anytime search (with delayed re-expansions as in ARA*). Experiments are conducted on a range of benchmark domains other than planning, using *Explicit Estimation Search* (rather than weighted A* as in RWA*) as the underlying search algorithm. In their experiments, restarting clearly outperforms continued search, yet the results of the repairing approach are moderately better than those of the restarting method.

10.3 Future Work

Throughout this thesis we have outlined ideas for future research that were specific to individual chapters. Here, we identify more general avenues of future work that apply to the thesis as a whole.

We have proposed using automatically extracted control knowledge in the form of landmarks to complement standard heuristics in planning. There are a number of ways in which this work can be extended. Firstly, we can try to improve the detection and usage of landmarks. For example, current landmark heuristics only account for the *first time* a landmark is achieved. Sometimes, however, landmarks have to be achieved and made false again several times in any plan. While methods exist for detecting the multiplicity of landmarks (Porteous and Cresswell, 2002; Zhu and Givan, 2003), it will be crucial to develop techniques for deriving orderings between the individual occurrences of such landmarks in order to exploit such landmarks effectively during planning.

In Chapter 9, we have given an outlook on finding and using *higher-order landmarks*, i. e., landmarks that are not based on the delete relaxation. We demonstrate that higher-order landmarks lead to significantly higher heuristic accuracy, yet the high computational cost for detecting and using such landmarks does not necessarily pay off. Additional research is needed to devise ways of using such landmarks efficiently during planning.

Going beyond landmarks, it would be interesting to investigate whether other forms of automatically generated control knowledge such as macro-actions (Fikes et al., 1972; Botea et al., 2004), symmetries (Fox and Long, 1999, 2002) etc. could be used in a similar fashion as landmarks to improve heuristic-search planning.

Manually generated control knowledge has been successfully used to guide planning systems (Bacchus and Kabanza, 2000; Doherty and Kvarnström, 2001). It is typically expressed with a rich syntax and used by systems supporting this syntax. For automatically extracted control

knowledge, the picture is different. Traditionally in the literature, each form of automatically extracted control knowledge has been proposed along with a unique way of exploiting it. For example, macro-actions have been compiled into the planning task, and symmetries have been supported via specific functionality in planning systems. One reason for this difference between the usage of manually generated control knowledge and automatically extracted control knowledge is that the former is "certain" and typically more complete. By contrast, automatically extracted control knowledge is sometimes not certain (i. e., not necessarily true), but may describe promising solution strategies. Hence, it is important to use automatically extracted control knowledge *in addition* to other successful planning techniques, rather than by itself. In particular, a planning system should still be effective if no control knowledge can be discovered for a given task.

It would be interesting to devise a suitable syntax for expressing various forms of automatically extracted control knowledge, and using this guidance in an automatic planner along with standard heuristics. DKEL (Haslum and Scholz, 2003) is a language that allows expressing several types of automatically extracted control knowledge such as macro-actions, invariants, irrelevant actions and replaceable action sequences. However, DKEL uses a special syntactic construct for each type of knowledge, without specifying any structural dependencies between the types. An important future development would be a language that is more amenable to describing connections between the different forms of knowledge. This might allow us to derive insights on how to transfer methods for finding and using control knowledge between the different types of knowledge.

One possibility for using such knowledge would be to extend our landmark heuristic and its usage of preferred operators to more general forms of control knowledge. Other possibilities include compiling control knowledge into heuristics, as Domshlak et al. (2010) have proposed for landmarks and abstraction heuristics. At the same time, novel methods for finding control knowledge could be devised, for example via search in the space of possible candidates given by a certain language. For such search, a *generate-test-refine* approach could be employed, as is commonly the case for invariant synthesis (Rintanen, 1998; Edelkamp and Helmert, 1999).

10.4 Closing Remarks

An important step towards realising intelligent systems capable of acting in real-world environments is the development of general techniques that quickly synthesise plans to achieve emerging objectives. Our thesis makes several contributions to the state of the art in that direction. Specifically we contribute: (1) knowledge-based planning heuristics and related automated discovery methods, (2) search procedures that can exploit knowledge-based heuristics, (3) anytime search procedures that can efficiently and consistently find high-quality plans, (4) an evaluation of various proposed search-enhancement strategies, and (5) a thorough empirical evaluation of our algorithmic contributions on a wide range of benchmark tasks from the community. Taken together, our contributions led to the automated planning system LAMA that won the satisficing track at IPC 2008.

Over the course of our work we have identified a number of interesting research agendas as outlined in the previous section. Going beyond these agendas, we hope that our work will contribute to advancing the use of automated planning techniques in real-world intelligent systems.

Bibliography

Sandip Aine, P. P. Chakrabarti, and Rajeev Kumar. AWA* – A window constrained anytime heuristic search algorithm. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2250–2255, 2007.

Anonymous. Diagnosis as planning: Two case studies, 2011. Submission to the *Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)* in progress. Author names witheld due to blind reviewing process.

Fahiem Bacchus. The AIPS'00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

J. Benton, Menkes van den Briel, and Subbarao Kambhampati. A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 34–41. AAAI Press, 2007.

Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

Mark Boddy, Johnathan Gohde, Tom Haigh, and Steven Harp. Course of action generation for cyber security using classical planning. In Susanne Biundo, Karen Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 12–21. AAAI Press, 2005.

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33, 2001.

Blai Bonet and Héctor Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. ICAPS 2003*, 2003.

Blai Bonet and Malte Helmert. Strengthening landmark heuristics via hitting sets. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 329–334. IOS Press, 2010.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Using component abstraction for automatic generation of macro-actions. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 181–190. AAAI Press, 2004.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Learning partial-order macros from solutions. In Susanne Biundo, Karen Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 231–240. AAAI Press, 2005.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

Daniel Bryce and Subbarao Kambhampati. A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28(1):47–83, 2007.

Olivier Buffet and Jörg Hoffmann. All that glitters is not gold: Using landmarks for reward shaping in FPG. In *Proceedings of the ICAPS 2010 Workshop on Planning and Scheduling Under Uncertainty*, 2010.

Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

Dunbo Cai, Jörg Hoffmann, and Malte Helmert. Enhancing the context-enhanced additive heuristic with precedence constraints. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 50–57. AAAI Press, 2009.

Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research*, 26:323–369, 2006.

Minh Binh Do and Subbarao Kambhampati. Sapa: A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research*, 20:155–194, 2003.

Minh Binh Do, Wheeler Ruml, and Rong Zhou. On-line planning and scheduling: An application to controlling modular printers. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1519–1523. AAAI Press, 2008.

Patrick Doherty and Jonas Kvarnström. TALplanner: A temporal logic based planner. *AI Magazine*, 22(3):95–102, 2001.

Carmel Domshlak, Michael Katz, and Sagi Lefler. When abstractions met landmarks. In Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz, editors, *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 50–56. AAAI Press, 2010.

Harry Dweighter. Elementary problem E2569. *The American Mathematical Monthly*, 82(10): 1010, 1975.

Stefan Edelkamp. Planning with pattern databases. In Amedeo Cesta and Daniel Borrajo, editors, *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, pages 13–24, Toledo, Spain, 2001.

Stefan Edelkamp. External symbolic heuristic search with pattern databases. In Susanne Biundo, Karen Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 51–60. AAAI Press, 2005.

Stefan Edelkamp and Malte Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In Susanne Biundo and Maria Fox, editors, *Recent Advances in AI Planning. 5th European Conference on Planning (ECP 1999)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pages 135–147, Heidelberg, 1999. Springer-Verlag.

Stefan Edelkamp and Malte Helmert. The model checking integrated planning system (MIPS). *AI Magazine*, 22(3):67–71, 2001.

Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th International Planning Competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2004.

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):65–88, 1995.

Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

Andreas Fink and Stefan Voß. Applications of modern heuristic search methods to pattern sequencing problems. *Computers and Operations Research*, 26(1):17–34, 1999.

Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 956–961. Morgan Kaufmann, 1999.

Maria Fox and Derek Long. Extending the exploitation of symmetries in planning. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, pages 83–91. AAAI Press, 2002.

Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Raquel Fuentetaja, Daniel Borrajo, and Carlos Linares López. A unified view of cost-based heuristics. In *ICAPS 2009 Workshop on Heuristics for Domain-Independent Planning*, pages 70–77, 2009.

David Furcy and Sven Koenig. Limited discrepancy beam search. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 125–131. Professional Book Center, 2005.

William H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Math*, 27:47–57, 1979.

Alfonso Gerevini and Ivan Serina. LPG: A planner based on local search for planning graphs with action costs. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, pages 13–22. AAAI Press, 2002.

Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5–6):619–668, 2009.

Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In Charles Rich and Jack Mostow, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 1998)*, pages 431–437. AAAI Press, 1998.

Eric A. Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

Eric A. Hansen, Shlomo Zilberstein, and Victor A. Danilchenko. Anytime heuristic search: First results. Technical Report CMPSCI 97-50, University of Massachusetts, Amherst, September 1997.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 1995)*, pages 607–615. Morgan Kaufmann, 1995.

Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to h^m. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 354–357. AAAI Press, 2009.

Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 140–149. AAAI Press, 2000.

Patrik Haslum and Ulrich Scholz. Domain knowledge in planning: Representation and use. In *Proceedings of the ICAPS-03 Workshop on PDDL*, 2003.

Malte Helmert. A planning heuristic based on causal graph analysis. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 161–170. AAAI Press, 2004.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Malte Helmert. *Understanding Planning Tasks – Domain Complexity and Heuristic Decomposition*, volume 4929 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2008.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

Malte Helmert. Landmark heuristics for the pancake problem. In Ariel Felner and Nathan Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 109–110. AAAI Press, 2010.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis,

editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169. AAAI Press, 2009.

Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 140–147. AAAI Press, 2008.

Malte Helmert and Silvia Richter. Fast downward — Making use of causal dependencies in the problem representation. Proceedings of the 2004 International Planning Competition, `http://www.tzi.de/~edelkamp/ipc-4/Proc/downward.pdf`, 2004.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 176–183. AAAI Press, 2007.

Malte Helmert, Minh Binh Do, and Ioannis Refanidis. IPC 2008, deterministic part. Web site, `http://ipc08.icaps-conference.org/`, 2008.

Jörg Hoffmann. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, 24:685–758, 2005.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Robert C. Holte. Common misconceptions concerning heuristic search. In Ariel Felner and Nathan Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 46–51. AAAI Press, 2010.

Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1–2):125–176, 1998.

Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1728–1733, 2009.

Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 174–181. AAAI Press, 2008a.

Michael Katz and Carmel Domshlak. Structural patterns heuristics via fork decomposition. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 182–189. AAAI Press, 2008b.

Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*, pages 359–363. John Wiley and Sons, 1992.

Henry A. Kautz and Joachim P. Walser. State-space planning by integer optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999)*, pages 526–533. AAAI Press, 1999.

Emil Keyder and Héctor Geffner. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, pages 588–592, 2008.

Emil Keyder and Héctor Geffner. Trees of shortest paths vs. Steiner trees: Understanding and improving delete relaxation heuristics. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1734–1749, 2009.

Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for and/or graphs. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 335–340. IOS Press, 2010.

Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.

Richard E. Korf. Depth-first iterative-deepending: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16 (NIPS 2003)*, 2004.

Maxim Likhachev, Dave Ferguson, Geoffrey J. Gordon, Anthony Stentz, and Sebastian Thrun. Anytime search in dynamic graphs. *Artificial Intelligence*, 172(14):1613–1643, 2008.

Alexandre Linhares and Horacio Hideki Yanasse. Connections between cutting-pattern sequencing, vlsi design and flexible machines. *Computers and Operations Research*, 29:1759–1772, 2002.

Nir Lipovetzky and Héctor Geffner. Inference and decomposition in planning using causal consistent chains. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors,

Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009). AAAI Press, 2009.

Yaxin Liu, Sven Koenig, and David Furcy. Speeding up the calculation of heuristics for heuristic search-based planning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI 2002)*, pages 484–491. AAAI Press, 2002.

Drew McDermott. The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55, 2000.

Vitaly Mirkis and Carmel Domshlak. Cost-sharing approximations for h^+. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 240–247. AAAI Press, 2007.

Allen Newell and Herbert A. Simon. GPS: A program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. Oldenbourg, 1963.

Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

Julie Porteous and Stephen Cresswell. Extending landmarks analysis to reason about resources and repetition. In *Proceedings of the 21st Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG '02)*, pages 45–54, 2002.

Julie Porteous, Laura Sebastia, and Jörg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In Amedeo Cesta and Daniel Borrajo, editors, *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, pages 37–48, Toledo, Spain, 2001.

Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 273–280. AAAI Press, 2009.

Silvia Richter and Matthias Westphal. The LAMA planner — Using landmark counting in heuristic search. IPC 2008 short papers, `http://ipc.informatik.uni-freiburg.de/Planners`, 2008.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 975–982. AAAI Press, 2008.

Silvia Richter, Jordan T. Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz, editors, *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 137–144. AAAI Press, 2010.

Bram Ridder and Derek Long. Integrating landmarks in partial order planners. In *ICAPS 2010 Doctoral Consortium*, 2010.

Jussi Rintanen. A planning algorithm not based on directional search. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR 1998)*, pages 617–624. Morgan Kaufmann, 1998.

Jussi Rintanen. Heuristics for planning with SAT. In Cohen David, editor, *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2010)*, volume 6308 of *Lecture Notes in Computer Science*, pages 414–428. Springer-Verlag, 2010.

Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, 2006.

Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz, editors, *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 246–249. AAAI Press, 2010.

Gabriele Röger, Patrick Eyerich, and Robert Mattmüller. TFD: A numeric temporal extension to Fast Downward. IPC 2008 short papers, `http://ipc.informatik.uni-freiburg.de/Planners`, 2008.

Wheeler Ruml and Minh Binh Do. Best-first utility-guided search. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2378–2384, 2007.

Laura Sebastia, Eva Onaindia, and Eliseo Marzal. Decomposition of planning problems. *AI Communications*, 19(1):49–81, 2006.

Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In William R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 2004)*, pages 440–446. AAAI Press, 1992.

Jordan Thayer and Wheeler Ruml. Anytime heuristic search: Frameworks and algorithms. In Ariel Felner and Nathan Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 121–128. AAAI Press, 2010.

Peter van Beek and Xinguang Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999)*, pages 585–590. AAAI Press, 1999.

Vincent Vidal. A lookahead strategy for heuristic search planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 150–159. AAAI Press, 2004.

Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi. Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In Ariel Felner and Nathan Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 100–107. AAAI Press, 2010.

Letao Wang, Jorge Baier, and Sheila McIlraith. Viewing landmarks as temporally extended goals. In *ICAPS 2009 Workshop on Heuristics for Domain-Independent Planning*, pages 49–56, 2009.

Uzi Zahavi, Ariel Felner, Jonathan Schaeffer, and Nathan Sturtevant. Inconsistent heuristics. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 1211–1216. AAAI Press, 2007.

Weixiong Zhang. Complete anytime beam search. In Charles Rich and Jack Mostow, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 1998)*, pages 425–430. AAAI Press, 1998.

Rong Zhou and Eric A. Hansen. Beam-stack search: Integrating backtracking with beam search. In Susanne Biundo, Karen Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 90–98. AAAI Press, 2005.

Lin Zhu and Robert Givan. Landmark extraction via planning graph propagation. In *ICAPS 2003 Doctoral Consortium*, pages 156–160, 2003.