

COUNTEREXAMPLE-GUIDED CARTESIAN
ABSTRACTION REFINEMENT AND SATURATED COST
PARTITIONING FOR OPTIMAL CLASSICAL PLANNING

Inauguraldissertation

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

JENDRIK SEIPP

aus Frankenberg (Eder), Deutschland

Basel, 2018

Originaldokument gespeichert auf dem Dokumentenserver der Universität Basel
edoc.unibas.ch

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Malte Helmert
Universität Basel, Dissertationsleiter, Fakultätsverantwortlicher

Prof. Dr. Jörg Hoffmann
Universität des Saarlandes, Korreferent

Basel, den 27.02.2018

Prof. Dr. Martin Spiess
Universität Basel, Dekan

To my parents.

ABSTRACT

Heuristic search with an admissible heuristic is one of the most prominent approaches to solving classical planning tasks optimally. We introduce a new family of admissible heuristics for classical planning, based on Cartesian abstractions, which we derive by counterexample-guided abstraction refinement. Since one abstraction usually is not informative enough for challenging planning tasks, we present several ways of creating diverse abstractions. To combine them admissibly, we introduce a new cost partitioning algorithm, which we call saturated cost partitioning. It considers the heuristics sequentially and uses the minimum amount of costs that preserves all heuristic estimates for the current heuristic before passing the remaining costs to subsequent heuristics until all heuristics have been served this way.

In Part II, we show that saturated cost partitioning is strongly influenced by the order in which it considers the heuristics. To find good orders, we present a greedy algorithm for creating an initial order and a hill-climbing search for optimizing a given order. Both algorithms make the resulting heuristics significantly more accurate. However, we obtain the strongest heuristics by maximizing over saturated cost partitioning heuristics computed for multiple orders, especially if we actively search for diverse orders.

Part III provides a theoretical and experimental comparison of saturated cost partitioning and other cost partitioning algorithms. Theoretically, we show that saturated cost partitioning dominates greedy zero-one cost partitioning. The difference between the two algorithms is that saturated cost partitioning opportunistically reuses unconsumed costs for subsequent heuristics. By applying this idea to uniform cost partitioning we obtain an opportunistic variant that dominates the original. We also prove that the maximum over suitable greedy zero-one cost partitioning heuristics dominates the canonical heuristic and show several non-dominance results for cost partitioning algorithms. The experimental analysis shows that saturated cost partitioning is the cost partitioning algorithm of choice in all evaluated settings and it even outperforms the previous state of the art in optimal classical planning.

ACKNOWLEDGMENTS

Writing this thesis would not have been possible without the support of many wonderful people to whom I am very grateful.

I would like to start by thanking Malte Helmert for being such an excellent supervisor and mentor. In the many years that I have known him, he has always been very eager to share his profound knowledge about planning and artificial intelligence with me. In addition, he has taught me so much about research in general, scientific writing and software engineering. Perhaps most importantly, his patience and humor have always made it a pleasure to work with him. Furthermore, I would like to thank Jörg Hoffmann for agreeing to be the second reviewer of this thesis. I also take this opportunity to thank my colleagues Salomé Eriksson, Patrick Ferber, Guillem Francès, Cedric Geissmann, Manuel Heusner, Thomas Keller, Florian Pommerening, Gabriele Röger, Silvan Sievers and Martin Wehrle for making the research group such an enjoyable place to work in.

I would also like to thank my friends from school, especially Markus Baumgartner, Christian Böhle, Christian Landeck, Oliver Neuschäfer, Karoline Schneider and Valérie Vogt. I am really glad that our friendship continues to feel as if we had never left school. I would also like to express my gratitude to my friends from university, especially Manuel Braun, Johannes Garimort, Philip Stahl and Jonas Sternisko, for the time we spent together in Freiburg and around Europe. And I would like to thank my friends and teammates from the Ultimate Frisbee teams Disconnection Freiburg and Freespeed Basel for the countless hours that we spent together on and off the field.

I owe a debt of gratitude to my sister Juliana, my brother Carsten and my extended family, especially Gabriele Ludwig, Jürgen Seipp, Luise and Karl Schubert, Anneliese and Kurt Seipp, and Klara and Hans Popp. Thank you for your kindness and support!

I will forever be grateful to my parents Adelheid and Reinhold for everything they have done for me. Thank you for being such awesome parents!

Last but definitely not least, I would like to express my gratitude to my girlfriend Andrea. Thank you for the wonderful time that we are having together and all the love and support that you are giving me!

CONTENTS

1	INTRODUCTION	1
1.1	Contributions	3
1.2	Experimental Setup	5
1.3	Publications	5
1.4	Awards	7
2	CLASSICAL PLANNING AND HEURISTICS	8
2.1	Planning Tasks	8
2.2	Transition Systems	9
2.3	Heuristics	11
3	COST PARTITIONING	15
3.1	Overview of Cost Partitioning Algorithms	16
3.2	Optimal Cost Partitioning	17
3.3	Post-hoc Optimization	19
3.4	Zero-One Cost Partitioning	20
3.5	Uniform Cost Partitioning	21
3.6	Canonical Heuristic	22
I	CARTESIAN ABSTRACTIONS	
4	CARTESIAN ABSTRACTION REFINEMENT	25
4.1	Cartesian Abstractions	25
4.2	Abstraction Refinement Algorithm	29
4.3	Implementation	36
4.4	Theoretical Runtime Analysis	39
4.5	Related Work	40
5	SATURATED COST PARTITIONING	43
5.1	Saturated Cost Function	44
5.2	Saturated Cost Partitioning Algorithm	45
5.3	Minimum Saturated Cost Function for Abstraction Heuristics	46
5.4	Interleaved Saturated Cost Partitioning Algorithm	50
6	MULTIPLE CARTESIAN ABSTRACTIONS	51
6.1	Diversification by Refinement Strategy	52
6.2	Abstraction by Goals	54
6.3	Abstraction by Landmarks	54
6.4	Abstraction by Landmarks: Improved	57
7	EXPERIMENTAL EVALUATION	59
7.1	Comparison of CEGAR to Other Abstraction Heuristics	59
7.2	Comparison of CEGAR Heuristics	62

II ORDERS FOR SATURATED COST PARTITIONING	
8 SINGLE ORDERS	66
8.1 Sparse Orders	67
8.2 Importance of Good Orders	69
8.3 Greedy Orders	71
8.4 Optimized Orders	75
8.5 Online Orders	78
9 MULTIPLE ORDERS	81
9.1 Diverse Orders	85
9.2 Summary of Improvements	89
III COMPARISON OF COST PARTITIONING ALGORITHMS	
10 THEORETICAL COMPARISON	91
10.1 Opportunistic Uniform Cost Partitioning	91
10.2 Dominances and Non-dominances	92
11 EXPERIMENTAL COMPARISON	98
11.1 Heuristic Orders	98
11.2 Abstraction Heuristics	100
11.3 Landmark Heuristics	108
11.4 Comparison of Different Approaches	109
IV CONCLUSION	
12 CONCLUSION	114
A APPENDIX	116
A.1 Pseudo-code for CEGAR Informed Transition Check	116
A.2 Detailed Results for Cost Partitioning Algorithms	119
BIBLIOGRAPHY	126

INTRODUCTION

Automated planning (Ghallab et al. 2004) is the problem of finding a sequence of actions that achieves a given goal. In this thesis, we focus on *classical* planning, where a single agent plans in a fully-observable world that only has actions with discrete and deterministic effects. This setting is still general enough to capture many interesting problems such as planning space missions, coordinating the transportation of goods, scheduling elevators, playing games like FreeCell or Tetris, or solving combinatorial puzzles.

Instead of trying to solve each of these different tasks with a specialized algorithm, we are interested in creating domain-independent planners that are able to handle all kinds of classical planning tasks with the same algorithm and without any domain-specific knowledge. Furthermore, instead of looking for any solution, we consider *optimal* classical planning, where only the cheapest among all solutions are accepted.

A* search (Hart et al. 1968) with an admissible heuristic (Pearl 1984) is one of the most prominent methods for optimal classical planning. Many of the strongest admissible heuristics are based on abstractions of the original planning task. Due to the way in which they simplify the original task, abstraction heuristics are guaranteed to be admissible. When creating an abstraction, we have to decide which parts of the original task to abstract away and which parts to preserve in more detail.

In Part I, we use counterexample-guided abstraction refinement (CEGAR) for this decision (Clarke et al. 2003). CEGAR is a well-known technique for model checking in large systems. It starts from a coarse abstraction and iteratively refines the abstraction in only the necessary places.

A key component of our approach is a new class of abstractions for classical planning, called *Cartesian abstractions*, which allow efficient and very fine-grained refinement. Cartesian abstractions are a generalization of the abstractions that underlie pattern database heuristics (Culberson and Schaeffer 1998; Edelkamp 2001) and domain abstraction (Hernádvölgyi and Holte 2000).

As the number of CEGAR iterations grows, one can observe diminishing returns: it takes more and more iterations to obtain further improvements in heuristic value. Therefore, we also show how to build multiple smaller *additive* abstractions instead of a single big one.

One way of composing admissible heuristics is to use the maximum of their estimates. This combination is always admissible if the component

heuristics are. *Cost partitioning* (Katz and Domshlak 2008; Yang et al. 2008) is an alternative that actually *combines* information from individual estimates instead of just selecting the most accurate one. It distributes operator costs among the heuristics, allowing to add up the heuristic estimates admissibly.

To combine multiple Cartesian abstraction heuristics admissibly, we introduce a new cost partitioning algorithm, which we call *saturated cost partitioning*. Saturated cost partitioning opportunistically exploits situations where some operator costs can be lowered without affecting the quality of the heuristic. Given an ordered sequence of abstractions, the saturated cost partitioning algorithm computes the smallest cost function $cost'$ that is sufficient for achieving the same heuristic values under the first abstraction as with the full cost function $cost$. It then assigns cost function $cost'$ to this abstraction and continues the process with the remaining abstractions, using the leftover costs $cost - cost'$ as the new overall cost function. Compared to other cost partitioning algorithms for abstraction heuristics, one advantage of saturated cost partitioning is that the abstractions can be created one at a time. In particular, no more than one abstraction needs to be held in memory simultaneously.

The simplest way to combine saturated cost partitioning with our CEGAR algorithm is to iteratively invoke the CEGAR loop on the same original task, only changing the cost functions between iterations to account for the costs that have already been assigned to previous abstractions. In our experiments this approach solves slightly fewer tasks compared to using a single Cartesian abstraction. This is because the resulting abstractions focus on mostly the same parts of the task. Therefore, we propose three methods for producing more *diverse* abstractions.

The first diversification method uses the fact that the CEGAR loop can often choose between multiple possibilities for refining the Cartesian abstraction and chooses a refinement based on the remaining costs. The second method computes abstractions for all goal atoms separately, while the third does so for all causal fact landmarks of the delete relaxation of the task (Keyder et al. 2010).

We evaluate Cartesian abstractions and saturated cost partitioning with and without these diversification strategies. Our results show that heuristics based on a single Cartesian abstraction are able to achieve competitive performance only in a few domains. However, constructing multiple abstractions in general and using landmarks to diversify the heuristics in particular leads to a significantly higher number of solved tasks and makes heuristics based on Cartesian abstractions outperform other state-of-the-art abstraction heuristics in many domains.

In Part II we investigate saturated cost partitioning in more detail. Since it assigns costs greedily, it is susceptible to the order in which it considers

the heuristics. Our analysis reveals that just changing the order of heuristics for saturated cost partitioning can make the difference between a perfect distance estimate and a highly inaccurate one. To find good orders, we propose two methods: a greedy algorithm and a hill climbing search in the space of all orders. The greedy orders and the optimized orders found by hill climbing significantly improve over random orders and we obtain the best results using optimized greedy orders. However, we show that it is often impossible to find a *single* order that provides good guidance across the state space: orders that are accurate for some states often turn out to be poor for others.

Maximizing over saturated cost partitioning heuristics for *multiple* orders allows us to use accurate heuristics for many different states and significantly improves over single-order heuristics. We show that our sets of saturated cost partitioning heuristics often contain heuristics that do not contribute any additional information during search. Therefore, we try to pick a subset of heuristics that complement each other by actively searching for multiple *diverse* orders.

Although there are many cost partitioning algorithms, the literature does not contain a thorough comparison. We provide such an analysis in Part III. Theoretically, we prove a number of dominance and non-dominance results, including the fact that saturated cost partitioning dominates greedy zero-one cost partitioning. It turns out that the key idea that distinguishes these two cost partitioning approaches can also be applied to uniform cost partitioning, leading to a new *opportunistic* version of uniform cost partitioning that dominates the original.

Experimentally, we report results for pattern databases that are derived systematically (Pommerening et al. 2013) and by hill climbing (Haslum et al. 2007), for Cartesian abstractions (Section 4.1), and for landmark heuristics (Karpas and Domshlak 2009), showing that saturated cost partitioning is the cost partitioning algorithm of choice for all considered kinds of heuristics on the IPC benchmarks.

1.1 Contributions

This thesis makes the following key contributions:

- In Chapter 4, we adapt counterexample-guided abstraction refinement from the model checking to the classical planning setting. Since existing classes of abstractions for classical planning, like pattern databases and merge-and-shrink abstractions, are not amenable to fine-grained, efficient abstraction refinement, we introduce a new class of abstractions for classical planning, called Cartesian abstrac-

tions. By iteratively refining Cartesian abstractions with CEGAR we obtain high-quality heuristics.

- Chapter 5 introduces a new cost partitioning algorithm, called saturated cost partitioning. The main idea of saturated cost partitioning is to give each heuristic only the fraction of remaining costs that the heuristic actually needs to justify its estimates and to save unconsumed costs for subsequent heuristics. We show that we can efficiently compute the minimum cost function that preserves all heuristic estimates of a given explicitly represented abstraction heuristic.
- In Chapter 6, we present two task decomposition methods: abstraction by goals and abstraction by landmarks. While these methods could be useful in many different scenarios, we use them to compute diverse Cartesian abstractions.
- In Chapter 7, we evaluate Cartesian abstraction heuristics derived by CEGAR experimentally, showing that saturated cost partitioning heuristics over Cartesian abstractions of the goals and landmarks task decompositions solve more tasks than existing abstraction heuristics in many benchmark domains.
- Chapter 8 shows that the order in which saturated cost partitioning considers the heuristics has a big impact on the quality of the resulting heuristic. We present two methods for computing good orders and show that both of them drastically improve the accuracy of saturated cost partitioning heuristics. We also demonstrate that we can often reduce the time and memory requirements of saturated cost partitioning heuristics by ignoring abstraction heuristics that do not contribute to the overall heuristic estimate. Furthermore, we show that a single order is not enough for obtaining an accurate heuristic for all states encountered during search.
- In Chapter 9, we solve this problem by maximizing over multiple saturated cost partitioning heuristics computed for different orders. This approach makes the resulting heuristics even more accurate, especially if we diversify the set of orders.
- Chapter 10 provides the first thorough theoretical comparison of cost partitioning algorithms. We show that saturated cost partitioning dominates greedy zero-one cost partitioning and propose an opportunistic version of uniform cost partitioning that dominates the original. In addition, we prove several other dominance and non-dominance results.

- In Chapter 11, we compare all cost partitioning algorithms experimentally using pattern databases, Cartesian abstraction heuristics and landmark heuristics. Our results show that saturated cost partitioning is the method of choice in all evaluated settings.

1.2 Experimental Setup

We use a common setup for all experiments in this thesis. The evaluated algorithms are implemented in the Fast Downward planning system (Helmert 2006) and the code is publicly available¹. We run experiments with the Downward Lab toolkit (Seipp et al. 2017). Our benchmark set² consists of all 1667 tasks from the optimization tracks of the 1998–2014 International Planning Competitions (IPC). The tasks belong to 40 different domains, some of which were used in several IPC instances. We do not filter duplicate tasks since this benchmark set is commonly used in the literature. We limit time and memory by 30 minutes and 3.5 GiB. The cluster on which we run our experiments consists of Intel Xeon E5-2660 processors with a clock speed of 2.2 GHz. All experimental data is published online³.

All planning tasks are given in the PDDL format (Fox and Long 2003) and we use the translator component of Fast Downward to convert them into SAS⁺ tasks (see Section 2.1). In our analyses we ignore the time taken for this conversion since it is the same for all compared algorithms.

Our main measure for comparing different heuristics h is the number of tasks solved by an A* search using h within the resource limits above. We often call this metric the *coverage score* or simply the *coverage* of an algorithm. To evaluate the accuracy of h , we often report the heuristic value for the initial state or the number of expansions A* makes when using h . For the latter number we ignore expansions on the last f layer, since their number depends on tie-breaking. Finally, we sometimes measure the evaluation speed of a heuristic by dividing the number of evaluated states by the time used for the A* search.

1.3 Publications

Some of the work in this thesis is based on the following publications:

Jendrik Seipp and Malte Helmert (2013). “Counterexample-guided Cartesian Abstraction Refinement.” In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*.

¹ Code: <https://doi.org/10.5281/zenodo.1204966>

² Benchmarks: <https://doi.org/10.5281/zenodo.1205019>

³ Experimental data: <https://doi.org/10.5281/zenodo.1204956>

- Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. AAAI Press, pp. 347–351.
- Jendrik Seipp and Malte Helmert (2014). “Diverse and Additive Cartesian Abstraction Heuristics.” In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. AAAI Press, pp. 289–297.
- Jendrik Seipp (2017). “Better Orders for Saturated Cost Partitioning in Optimal Classical Planning.” In: *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*. Ed. by Alex Fukunaga and Akihiro Kishimoto. AAAI Press, pp. 149–153.
- Jendrik Seipp, Thomas Keller, and Malte Helmert (2017a). “A Comparison of Cost Partitioning Algorithms for Optimal Classical Planning.” In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*. AAAI Press, pp. 259–268.
- Jendrik Seipp, Thomas Keller, and Malte Helmert (2017b). “Narrowing the Gap Between Saturated and Optimal Cost Partitioning for Classical Planning.” In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*. AAAI Press, pp. 3651–3657.

The following papers were published during my doctoral studies but are not covered by this thesis:

- Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp (2015). “From Non-Negative to General Operator Cost Partitioning.” In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*. AAAI Press, pp. 3335–3341.
- Jendrik Seipp, Florian Pommerening, and Malte Helmert (2015a). “New Optimization Functions for Potential Heuristics.” In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. Ed. by Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, pp. 193–201.
- Jendrik Seipp, Silvan Sievers, Malte Helmert, and Frank Hutter (2015b). “Automatic Configuration of Sequential Planning Portfolios.” In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*. AAAI Press, pp. 3364–3370.
- Thomas Keller, Florian Pommerening, Jendrik Seipp, Florian Geißer, and Robert Mattmüller (2016). “State-dependent Cost Partitionings for Cartesian Abstractions in Classical Planning.” In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. Ed. by Subbarao Kambhampati. AAAI Press, pp. 3161–3169.
- Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert (2016). “Correlation Complexity of Classical Planning Domains.” In: *Proceedings of the 25th International Joint Conference on Artificial Intel-*

ligence (IJCAI 2016). Ed. by Subbarao Kambhampati. AAAI Press, pp. 3242–3250.

1.4 Awards

Work that is part of this thesis or closely related received the following awards:

- **AAAI 2015 Outstanding Paper Award** for the paper *From Non-Negative to General Operator Cost Partitioning* with Florian Pommerening, Malte Helmert and Gabriele Röger at the 29th AAAI Conference on Artificial Intelligence (AAAI 2015).
- **Winner, Unsolvability IPC 2016** for the planning system *Fast Downward Aidos* with Florian Pommerening, Silvan Sievers, Martin Wehrle, Chris Fawcett and Yusra Alkhazraji at the 1st Unsolvability International Planning Competition (UIPC 2016) at ICAPS 2016.
- **SoCS 2017 Best Student Paper Award** for the paper *Better Orders for Saturated Cost Partitioning in Optimal Classical Planning* at the 10th Annual Symposium on Combinatorial Search (SoCS 2017).

We begin by formally introducing planning tasks, transition systems and heuristics.

2.1 Planning Tasks

In the first part of this thesis, we use a toy planning task as a running example. The task is adapted from the Gripper domain (McDermott 2000) in which a robot has to transport balls from room A to room B . In our example task the robot has a single gripper G and there is only one ball. The robot can grab and drop the ball and move between the two rooms. Initially, the robot is in room A , so assuming all operators cost 1, the cheapest solution for this task is to let the robot grab the ball in room A , move to room B and drop the ball there.

To formalize classical planning problems such as this example, we use a SAS⁺-like (Bäckström and Nebel 1995) representation.

Definition 2.1 Planning tasks.

A planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where:

- $\mathcal{V} = \langle v_1, \dots, v_n \rangle$ is a finite sequence of state variables, each with an associated finite domain $\text{dom}(v_i)$.

An atom is a pair $\langle v, d \rangle$, also written $v \mapsto d$, with $v \in \mathcal{V}$ and $d \in \text{dom}(v)$.

A partial state s is an assignment that maps a subset $\text{vars}(s)$ of \mathcal{V} to values in their respective domains. We write $s[v] \in \text{dom}(v)$ for the value which s assigns to the variable v . Partial states defined on all variables are called states, and $S(\Pi)$ is the set of all states of Π . We will interchangeably treat partial states as mappings from variables to values or as sets of atoms.

The update of partial state s with partial state t , written $s \oplus t$, is the partial state with $\text{vars}(s \oplus t) = \text{vars}(s) \cup \text{vars}(t)$, $(s \oplus t)[v] = t[v]$ for all $v \in \text{vars}(t)$, and $(s \oplus t)[v] = s[v]$ for all $v \in \text{vars}(s) \setminus \text{vars}(t)$.

- \mathcal{O} is a finite set of operators. Each operator o has a precondition $\text{pre}(o)$, an effect $\text{eff}(o)$ and a non-negative cost $\text{cost}(o) \in \mathbb{R}_0^+$. The precondition $\text{pre}(o)$ and effect $\text{eff}(o)$ are partial states. The postcondition $\text{post}(o)$ of an operator o is defined as $\text{pre}(o) \oplus \text{eff}(o)$. An operator $o \in \mathcal{O}$ is applicable in state s if $\text{pre}(o) \subseteq s$. Applying o in s results in the state $s[o] = s \oplus \text{eff}(o)$.
- $s_0 \in S(\Pi)$ is the initial state and s_\star is a partial state, called the goal.

Variables $\mathcal{V} = \langle robot, ball \rangle$, $dom(robot) = \{A, B\}$, $dom(ball) = \{A, B, G\}$

Operators $\mathcal{O} = \{\text{move-A-B}, \text{move-B-A}, \text{grab-in-A}, \text{grab-in-B}, \text{drop-in-A}, \text{drop-in-B}\}$

- $pre(\text{move-A-B}) = \{robot \mapsto A\}$, $eff(\text{move-A-B}) = \{robot \mapsto B\}$,
 $cost(\text{move-A-B}) = 1$.
- $pre(\text{move-B-A}) = \{robot \mapsto B\}$, $eff(\text{move-B-A}) = \{robot \mapsto A\}$,
 $cost(\text{move-B-A}) = 1$.
- $pre(\text{grab-in-A}) = \{robot \mapsto A, ball \mapsto A\}$, $eff(\text{grab-in-A}) = \{ball \mapsto G\}$,
 $cost(\text{grab-in-A}) = 1$.
- $pre(\text{grab-in-B}) = \{robot \mapsto B, ball \mapsto B\}$, $eff(\text{grab-in-B}) = \{ball \mapsto G\}$,
 $cost(\text{grab-in-B}) = 1$.
- $pre(\text{drop-in-A}) = \{robot \mapsto A, ball \mapsto G\}$, $eff(\text{drop-in-A}) = \{ball \mapsto A\}$,
 $cost(\text{drop-in-A}) = 1$.
- $pre(\text{drop-in-B}) = \{robot \mapsto B, ball \mapsto G\}$, $eff(\text{drop-in-B}) = \{ball \mapsto B\}$,
 $cost(\text{drop-in-B}) = 1$.

Initial State $s_0(robot) = A$ and $s_0(ball) = A$ (or $s_0 = \langle A, A \rangle$)

Goal $s_* = \{ball \mapsto B\}$

Figure 2.1: Definition of the example Gripper task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ with a single ball, a gripper G and two rooms A and B .

Figure 2.1 shows how the example Gripper task can be formalized using this definition. We will frequently use a tuple notation for states and write $\langle d_1, \dots, d_n \rangle$ to denote the state $\{v_1 \mapsto d_1, \dots, v_n \mapsto d_n\}$. (Facilitating such a notation is the main reason why we define \mathcal{V} as a sequence rather than a set.)

2.2 Transition Systems

The notion of transition systems is central for assigning semantics to planning tasks:

Definition 2.2 Transition Systems.

A transition system \mathcal{T} is a directed, labeled graph $\mathcal{T} = \langle S, \mathcal{L}, T, s_0, S_* \rangle$, where S is a finite set of states; \mathcal{L} is a finite set of labels; T is a set of labeled transitions $s \xrightarrow{l} s'$ for $s, s' \in S$ and $l \in \mathcal{L}$; $s_0 \in S$ is the initial state; and $S_* \subseteq S$ is the set of goal states.

We write $S(\mathcal{T})$ for the states in \mathcal{T} , $\mathcal{L}(\mathcal{T})$ for the labels in \mathcal{T} , $T(\mathcal{T})$ for the transitions in \mathcal{T} and $S_\star(\mathcal{T})$ for the goal states in \mathcal{T} .

Definition 2.3 Cost Functions.

A cost function for transition system \mathcal{T} is a function $cost : \mathcal{L}(\mathcal{T}) \rightarrow \mathbb{R} \cup \{-\infty\}$. We write $\mathcal{C}(\mathcal{T})$ for the set of all cost functions for \mathcal{T} .

A cost function $cost$ is called non-negative if $cost(l) \geq 0$ for all labels l . Unrestricted cost functions are also called possibly negative.

We only consider non-negative cost functions as inputs to cost partitioning algorithms, but following Pommerening et al. (2015), we permit possibly negative cost functions within cost partitionings. For brevity, we sometimes use a tuple notation for cost functions and write $cost = \langle cost(l_1), \dots, cost(l_k) \rangle$.

Definition 2.4 Fixed-cost Transition Systems.

A fixed-cost transition system is a pair $\langle \mathcal{T}, cost \rangle$, where \mathcal{T} is a transition system and $cost$ is a cost function for \mathcal{T} . We also refer to transition systems as parameterized-cost transition systems to emphasize the distinction to fixed-cost transition systems.

A fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ is the weighted digraph obtained from \mathcal{T} by assigning all transitions $s \xrightarrow{l} s' \in T(\mathcal{T})$ the weight $cost(l)$. We distinguish between fixed-cost and parameterized-cost transition systems since for cost partitioning we need to be able to use different cost functions for the same transition system. A fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ is regular if $cost$ is non-negative.

Definition 2.5 Goal Distance.

For a fixed-cost transition system $\langle \mathcal{T}, cost \rangle$, the goal distance $h_{\mathcal{T}, cost}^*(s) \in \mathbb{R} \cup \{-\infty, \infty\}$ of a state $s \in S(\mathcal{T})$ is the cost of a cheapest path weighted by $cost$ from s to a goal state in $S_\star(\mathcal{T})$. The goal distance is ∞ if no such path exists and $-\infty$ if paths of arbitrarily low negative cost exist.

For a parameterized-cost transition system \mathcal{T} , the goal distance of state $s \in S(\mathcal{T})$ is defined as $h_{\mathcal{T}}^*(s, cost) = h_{\mathcal{T}, cost}^*(s)$.

Note that for regular fixed-cost transition systems $\langle \mathcal{T}, cost \rangle$, we have $h_{\mathcal{T}, cost}^*(s) \geq 0$ for all states $s \in S(\mathcal{T})$.

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ with operator cost function $cost$ induces a fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ with states $S(\Pi)$, labels \mathcal{O} , transitions $\{s \xrightarrow{o} (s \oplus \text{eff}(o)) \mid s \in S(\Pi), o \in \mathcal{O}, \text{pre}(o) \subseteq s\}$, initial state s_0 and goal states $\{s \in S(\Pi) \mid s_\star \subseteq s\}$.

Definition 2.6 Regression, plans and traces.

Let $\langle \mathcal{T}, cost \rangle$ be a fixed-cost transition system.

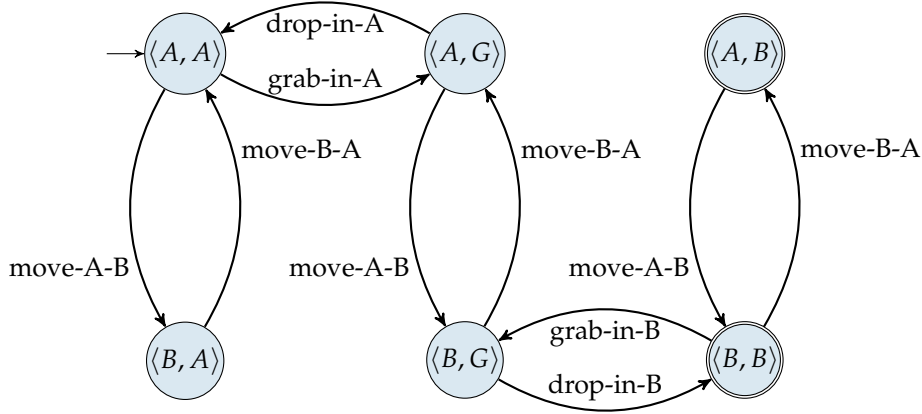


Figure 2.2: Transition system for the example Gripper task with a single ball, a gripper G and two rooms A and B .

The regression of a set of states $S' \subseteq S(\mathcal{T})$ with respect to a label $l \in \mathcal{L}(\mathcal{T})$ is defined as $\text{regr}(S', l) = \{s \in S(\mathcal{T}) \mid s \xrightarrow{l} s' \in T(\mathcal{T}) \wedge s' \in S'\}$.

A sequence of transitions $\langle s^0 \xrightarrow{l_1} s^1, s^1 \xrightarrow{l_2} s^2, \dots, s^{k-1} \xrightarrow{l_k} s^k \rangle$ is called a trace from s^0 to s^k . The empty sequence is considered a trace from s to s for all states s . A trace from state s to some state $s' \in S_*$ is called a goal trace for s . The sequence of labels $\langle l_1, l_2, \dots, l_k \rangle$ of a goal trace for state s is called a plan for s . We often write “goal trace” and “plan” instead of “goal trace for s_0 ” and “plan for s_0 ”.

A plan is *optimal* if the sum of assigned weights along the path is minimal. Optimal planning is the following problem: given a planning task Π with initial state s_0 and operator cost function cost , find an optimal plan starting in s_0 in the transition system induced by Π and weighted by cost , or prove that no plan starting in s_0 exists. Figure 2.2 shows the (unweighted) transition system induced by the Gripper example. We follow the common convention and always mark the initial state with an unlabeled incoming edge and goal states with double borders.

2.3 Heuristics

Heuristics estimate the cost of a cheapest path between a given state and the nearest goal state (Pearl 1984). We define two types of heuristics: a *parameterized-cost* heuristic is a function from states and cost functions to cost estimates. In contrast, a *fixed-cost* heuristic is defined as a function only from states to cost estimates.

Definition 2.7 Heuristics.

Let \mathcal{T} be a parameterized-cost transition system. A parameterized-cost heuristic for \mathcal{T} is a function $h : S(\mathcal{T}) \times \mathcal{C}(\mathcal{T}) \rightarrow \mathbb{R} \cup \{-\infty, \infty\}$.

Let $\langle \mathcal{T}, cost \rangle$ be a fixed-cost transition system. A fixed-cost heuristic for \mathcal{T} is a function $h : S(\mathcal{T}) \rightarrow \mathbb{R} \cup \{-\infty, \infty\}$.

Since almost all heuristics in this thesis are parameterized-cost heuristics, we usually simply call them *heuristics*.

Definition 2.8 Properties of Heuristics.

Let \mathcal{T} be a parameterized-cost transition system and let h be a parameterized-cost heuristic for \mathcal{T} . Then

- h is goal-aware if $h(s, cost) \leq 0$ for all cost functions $cost \in \mathcal{C}(\mathcal{T})$ and goal states $s \in S_*(\mathcal{T})$,
- h is consistent if $h(s, cost) \leq cost(o) + h(s[o], cost)$ for all cost functions $cost \in \mathcal{C}(\mathcal{T})$ and states $s \in S(\mathcal{T})$,
- h is admissible if $h(s, cost) \leq h_{\mathcal{T}}^*(s, cost)$ for all cost functions $cost \in \mathcal{C}(\mathcal{T})$ and states $s \in S(\mathcal{T})$, and
- h is cost-monotonic if $h(s, cost') \leq h(s, cost)$ for all states $s \in S(\mathcal{T})$ whenever $cost' \leq cost$.

Goal-awareness, consistency and admissibility are defined analogously for fixed-cost heuristics. A goal-aware and consistent heuristic is also admissible (Russell and Norvig 1995). Using an admissible heuristic in an A^* search yields optimal solutions (Hart et al. 1968).

Cost-monotonicity In less technical terms, making transitions more expensive cannot decrease heuristic estimates of cost-monotonic heuristics. Examples for cost-monotonic heuristics from the literature include the perfect heuristic h^* , the optimal delete-relaxation heuristic h^+ (Hoffmann and Nebel 2001), the family of critical-path heuristics h^C (Haslum 2012) and consequently the h^m heuristic family (Haslum and Geffner 2000), which is a special case of h^C . In contrast, the h^{LM-cut} (Helmert and Domshlak 2009) and h^{FF} (Hoffmann and Nebel 2001) heuristics are not cost-monotonic due to their use of tie-breaking.

Label l affects heuristic h if heuristic estimates of h may depend on $cost(l)$, i.e., there exist states s and non-negative cost functions $cost$ and $cost'$ which differ only on l with $h(s, cost) \neq h(s, cost')$. We define $\mathcal{A}(h) = \{l \in \mathcal{L} \mid l \text{ affects } h\}$. Heuristics h and h' with $\mathcal{A}(h) \cap \mathcal{A}(h') = \emptyset$ are called *independent*.

By losing some distinctions between states, we can create an *abstraction* of a planning task. This allows us to obtain a more “coarse-grained”, and hence smaller, transition system. For this thesis, it is convenient to use a definition based on equivalence relations:

Definition 2.9 Abstractions.

Let Π be a planning task inducing the transition system $\langle S, \mathcal{L}, T, s_0, S_\star \rangle$.

An abstraction relation \sim for Π is an equivalence relation on S . Its equivalence classes are called abstract states. We write $[s]_\sim$ for the equivalence class to which s belongs. The function mapping s to $[s]_\sim$ is called the abstraction function. We omit the subscript \sim where clear from context.

The abstract transition system induced by \sim is the transition system \mathcal{T}' with states $\{[s] \mid s \in S\}$, labels \mathcal{L} , transitions $\{[s] \xrightarrow{l} [s'] \mid s \xrightarrow{l} s' \in T\}$, initial state $[s_0]$ and goal states $\{[s] \mid s \in S_\star\}$. We call $\mathcal{T}' = \langle S', \mathcal{L}', T', s'_0, S'_\star \rangle$ an induced abstraction. Enlarging S', \mathcal{L}', T' or S'_\star turns \mathcal{T}' into a (non-induced) abstraction.

Induced abstractions are also called *strict homomorphisms*, and general abstractions are called *homomorphisms*. We generally assume abstractions to be induced and will make it explicit when we speak about non-induced abstractions.

In the context of an abstraction, the planning task Π on which the abstract transition system is based is called the *concrete* task. Similarly, we will speak of concrete states, concrete transitions etc. to distinguish them from abstract ones.

Abstraction Heuristics We can use abstraction to define admissible (and consistent) heuristics for planning (e.g., Helmert et al. 2007; Katz and Domshlak 2008, 2010). We write \mathcal{T}_h for the abstract transition system that *abstraction heuristic* h is based on. The heuristic estimate $h(s, cost)$ of an abstraction heuristic h is defined as the cost of an optimal plan starting from $[s]$ in abstract transition system \mathcal{T}_h weighted by $cost$, i.e., $h_{\mathcal{T}_h, cost}^*([s])$, or ∞ if no plan starting from $[s]$ exists. Abstraction heuristics are admissible since abstraction preserves paths, i.e., all paths in a concrete transition system are also present in an induced abstraction. They are also cost-monotonic since increasing the weight of transitions can only increase goal distances. Practically useful abstractions should be efficiently computable and give rise to informative heuristics. These are conflicting objectives. We compare different classes of abstraction heuristics and show examples in Section 4.1.

Landmarks Another important concept from the literature that we use in this thesis are *landmarks* (e.g., Karpas and Domshlak 2009; Richter and Westphal 2010). A *fact landmark* for a state s is an atom that has to be true at least once in all plans for s . We use fact landmarks in Section 6.3 to decompose a given task into multiple subtasks.

A *disjunctive action landmark* for a state s is a set of operators $L \subseteq \mathcal{O}$ for which we have that at least one operator $o \in L$ must be part of any plan for s . For each fact landmark $v \mapsto d$, the set of operators that achieve $v \mapsto d$ is a disjunctive action landmark.

Landmark Heuristics For this thesis, it is useful to view a *landmark heuristic* for a state s and a (single) disjunctive action landmark L for s as a two-state (non-induced) abstraction heuristic with the abstract initial state “ L not achieved” and the abstract goal state “ L achieved”. For all operators in L there is a transition from “ L not achieved” to “ L achieved” and s is mapped to “ L not achieved”. Since the set of landmarks that have to be achieved changes between states, the set of landmark heuristics also differs from state to state.

COST PARTITIONING

For finding solutions in huge transition systems, it can be beneficial to use multiple heuristics that focus on different parts of the state space (e.g., Holte et al. 2006). The question is how we can combine the heuristics so that the resulting heuristic is both informative and admissible. Maximizing over the heuristic estimates in each state guarantees admissibility if each component heuristic is admissible, but the resulting heuristic is only as strong as the strongest component heuristic in each state.

In contrast, *cost partitioning* combines the information contained in the component heuristics and yields a heuristic that is often much stronger than any of its components. It preserves admissibility by distributing the costs of a task among the component heuristics (Katz and Domshlak 2008). Cost partitioning is more general than maximizing over multiple heuristics, since we can express any maximization by a cost partitioning that uses all costs for the heuristic with the highest estimate. Following Pommerening et al. (2015), we allow negative costs in cost partitionings.

Definition 3.1 Cost Partitioning.

Let $\mathcal{H} = \langle h_1, \dots, h_n \rangle$ be a tuple of admissible parameterized-cost heuristics for a regular fixed-cost transition system $\langle \mathcal{T}, cost \rangle$. A cost partitioning for cost and \mathcal{H} is a tuple $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$ of possibly negative cost functions whose sum is bounded by cost: $\sum_{i=1}^n cost_i(l) \leq cost(l)$ for all $l \in \mathcal{L}(\mathcal{T})$. The cost-partitioned fixed-cost heuristic $h^{\mathcal{C}}$ is defined as $h^{\mathcal{C}}(s) = \sum_{i=1}^n h_i(s, cost_i)$. If any term in the sum is ∞ the sum is defined as ∞ , even if another term is $-\infty$.

Intuitively, since each heuristic yields an admissible estimate, their sum is also admissible due to the way \mathcal{C} divides the costs among the heuristics.

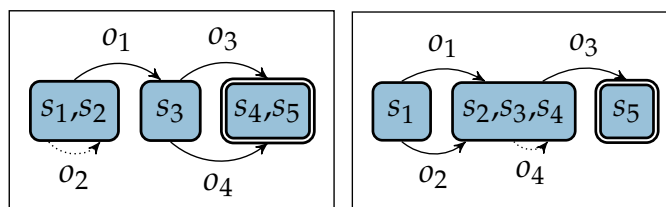


Figure 3.1: Example abstraction heuristics. The cost function is $cost = \langle 4, 1, 4, 1 \rangle$, i.e., operators o_1 and o_3 cost 4, whereas o_2 and o_4 cost 1. In all figures depicting abstraction heuristics, abstract states are rounded rectangles and abstract self-loops are dotted.

We illustrate the concept of cost partitioning with the two abstraction heuristics h_1 and h_2 and cost function $cost = \langle 4, 1, 4, 1 \rangle$ in Figure 3.1. We have $h_1(s_1) = 5$ and $h_2(s_1) = 5$ under $cost$. Therefore, maximizing over the two estimates also yields a heuristic value of 5. We show below that a suitable cost partitioning is able to raise the resulting heuristic value significantly. For example, the cost partitioning $\mathcal{C} = \langle cost_1, cost_2 \rangle$ with $cost_1 = \langle 4, 0, 1, 1 \rangle$ and $cost_2 = \langle 0, 0, 3, 0 \rangle$ yields $h^{\mathcal{C}}(s_1) = 5 + 3 = 8$.

3.1 Overview of Cost Partitioning Algorithms

Before the emergence of cost partitioning methods, additive admissible heuristics such as *disjoint pattern databases* (PDBs) exploited the natural independence between multiple heuristics that arises when no operator contributes to the estimate of more than one heuristic (Edelkamp 2006; Felner et al. 2004; Korf and Felner 2002). The further development of this idea led to the *canonical heuristic* for pattern databases (Haslum et al. 2007), which computes all maximal subsets of pairwise independent abstractions and then uses the maximum of all sums over independent abstractions as the heuristic value. In a formal sense, this is the most accurate heuristic that can be derived from a given set of heuristics if the only available information apart from the heuristic values is which pairs of heuristics are independent.

Zero-one cost partitioning was introduced as a generalization of such independence-based additive heuristics. This approach, first formally described by Haslum et al. (2005), artificially enforces independence between heuristics by treating operators as free of cost if they have already been “consumed” by another heuristic.

The development of cost partitioning as a general concept is due to Katz and Domshlak (2008), who also introduced *uniform cost partitioning* as a practical cost partitioning method: instead of assigning the whole cost of an operator to a single heuristic, the cost is equally distributed among all heuristics for which the operator is relevant.

All these approaches are theoretically dominated by *optimal cost partitioning*, which has been shown to be computable in polynomial time for abstraction (Katz and Domshlak 2008, 2010) and landmark (Karpas and Domshlak 2009) heuristics and has more recently been extended to permit negative costs (Pommerening et al. 2015).

However, these promising theoretical results do (so far) not translate well into practice: Pommerening et al. (2013) showed that even for comparatively small pattern database heuristics, optimal cost partitioning tends to be prohibitively expensive, as it requires to solve linear programs (LPs) with “up to millions of variables and billions of constraints for realistic problem sizes” (Pommerening et al. 2013). Their *post-hoc optimization*

heuristic is an approximation to optimal cost partitioning based on a linear program where a single weight is computed for each heuristic, which dramatically reduces the solution space, making the approach much cheaper but less accurate than optimal cost partitioning.

The latest addition to the collection of cost partitioning algorithms is *delta cost partitioning* (Fan et al. 2017): given an original cost function (with possibly diverse operator costs), it computes a set of cost functions where each cost function only maps to at most two different values. Using these cost functions instead of a single cost function with diverse costs is beneficial for label reduction (Sievers et al. 2014) in merge-and-shrink abstractions (Helmert et al. 2014), since only labels mapped to the same cost can potentially be combined. However, the applicability of delta cost partitioning is limited. Since the resulting number of cost functions depends on the original cost function, we cannot use it for a fixed number of heuristics. For example, given a unit cost task, the algorithm produces a cost partitioning that only consists of the original cost function. We therefore do not analyze delta cost partitioning further in this thesis.

In the remainder of this chapter, we formally define the other cost partitioning algorithms introduced above.

3.2 Optimal Cost Partitioning

An optimal cost partitioning for a given state is a cost partitioning \mathcal{C}^* where $h^{\mathcal{C}^*}(s)$ is maximal among all possible cost partitionings.

Definition 3.2 Optimal Cost Partitioning.

Let $\mathcal{H} = \langle h_1, \dots, h_n \rangle$ be a tuple of admissible heuristics for a regular fixed-cost transition system $\langle \mathcal{T}, cost \rangle$. An optimal cost partitioning for cost, \mathcal{H} and a state $s \in S(\mathcal{T})$ is a cost partitioning \mathcal{C}^* where $h^{\mathcal{C}^*}(s)$ is maximal among all cost partitionings for cost and \mathcal{H} . The optimal cost partitioning heuristic h^{OCP} is a parameterized-cost heuristic defined as $h^{OCP}(s, cost) = h^{\mathcal{C}^*}(s)$, where \mathcal{C}^* is an optimal cost partitioning for cost, \mathcal{H} and state $s \in S(\mathcal{T})$.

Whether an optimal cost partitioning can be computed depends on the type of heuristics. The cases that are best understood are abstraction and landmark heuristics.

Optimal Cost Partitioning for Abstraction Heuristics Katz and Domshlak (2008, 2010) presented an algorithm for computing optimal cost partitionings over explicitly represented abstraction heuristics. Since it is based on linear programming, the algorithm runs in polynomial time. Figure 3.2 shows one possible formalization of their linear program. Given a state s , a cost function $cost$ and a tuple of abstraction heuristics \mathcal{H} , the linear pro-

Maximize $\sum_{h \in \mathcal{H}} H_{[s]_h}^h$ subject to

$$H_a^h \leq 0 \quad \text{for all } h \in \mathcal{H} \text{ and } a \in S_*(\mathcal{T}_h) \quad (3.1)$$

$$H_a^h - H_b^h \leq C_l^h \quad \text{for all } h \in \mathcal{H} \text{ and } a \xrightarrow{l} b \in T(\mathcal{T}_h) \quad (3.2)$$

$$\sum_{h \in \mathcal{H}} C_l^h \leq \text{cost}(l) \quad \text{for all } l \in \mathcal{L}(\mathcal{T}_h) \quad (3.3)$$

Figure 3.2: Linear program that computes an optimal cost partitioning for a regular concrete fixed-cost transition system $\langle \mathcal{T}, \text{cost} \rangle$, a tuple of abstraction heuristics $\mathcal{H} = \langle h_1, \dots, h_n \rangle$ for \mathcal{T} and a concrete state $s \in S(\mathcal{T})$. Each abstraction heuristic $h \in \mathcal{H}$ is associated with an abstract transition system \mathcal{T}_h and an abstraction function that maps each state $s \in S(\mathcal{T})$ to the corresponding abstract state $[s]_h \in S(\mathcal{T}_h)$. The tuple $\langle \text{cost}_1, \dots, \text{cost}_n \rangle$, where $\text{cost}_i(l) = C_l^{h_i}$, is an optimal cost partitioning.

gram maximizes the sum of abstract goal distances of the abstract states that s is mapped to by the abstraction functions.

For each abstraction heuristic h with associated abstract transition system \mathcal{T}_h , the linear program contains variables H_a^h for each abstract state $a \in S(\mathcal{T}_h)$ and C_l^h for each label $l \in \mathcal{L}(\mathcal{T}_h)$. Variable C_l^h encodes the cost of label l in \mathcal{T}_h . Together with the optimization function, constraints 3.1 and 3.2 ensure that each variable H_a^h encodes the goal distance of abstract state $a \in S(\mathcal{T}_h)$ under the cost function encoded by C^h . Constraint 3.3 checks that the encoded cost functions form a cost partitioning.

Example For the abstraction heuristics h_1 and h_2 in Figure 3.1 the example cost partitioning \mathcal{C} from above is an optimal cost partitioning: $\mathcal{C} = \langle \text{cost}_1, \text{cost}_2 \rangle$ with $\text{cost}_1 = \langle 4, 0, 1, 1 \rangle$ and $\text{cost}_2 = \langle 0, 0, 3, 0 \rangle$ yields $h^{\mathcal{C}}(s_1) = 5 + 3 = 8$. However, there are multiple optimal solutions for the LP: the cost partitioning $\mathcal{C}' = \langle \text{cost}_1, \text{cost}_2 \rangle$ with $\text{cost}_1 = \langle 3.5, 0, 1, 1 \rangle$ and $\text{cost}_2 = \langle 0.5, 1, 3, 0 \rangle$ also yields $h^{\mathcal{C}'}(s_1) = 4.5 + 3.5 = 8$.

Computing optimal cost partitionings for some or even all states encountered during search has been shown to be a practically viable approach for landmark heuristics and certain classes of implicit abstraction heuristics (Karpas and Domshlak 2009; Karpas et al. 2011; Katz and Domshlak 2010). However, despite the promising theoretical guarantees, computing optimal cost partitionings can already be prohibitively expensive for explicit abstractions of modest size: Pommerening et al. (2013), for example, performed experiments with systematic pattern databases of up to size 2, showing that for 249 out of 1396 tasks in their benchmark set, computing

an optimal cost partitioning for a single state is infeasible even with a 24-hour time limit and a 2 GiB memory limit. This includes 206 cases where the LP computation runs out of memory and 43 timeouts after 24 hours. We give further evidence for the impractical time and memory requirements of optimal cost partitioning in Section 11.2.

Consequently, several alternatives to optimal cost partitioning with varying time vs. accuracy tradeoffs have been proposed. We remark that even for suboptimal cost partitionings the resulting cost-partitioned heuristics remain admissible (since we only allow admissible component heuristics). Therefore, all cost-partitioned heuristics are guaranteed to find optimal solutions when used in an A^* search.

3.3 Post-hoc Optimization

Like optimal cost partitioning, *post-hoc optimization* (Pommerening et al. 2013) is a cost partitioning method based on linear programming. Each component heuristic is assigned a single real-valued weight in the range $[0, 1]$, and the overall heuristic value is the weighted sum of component heuristics. For each label, there is a constraint that ensures that the total weight of all heuristics affected by the label sum up to at most 1. A solution of the post-hoc optimization LP corresponds to a cost partitioning where operators that do not affect a given heuristic are assigned a cost of 0, and operators affecting a heuristic which receives the weight w_i are assigned the fraction w_i of their full cost.

Definition 3.3 Post-hoc Optimization.

Let $\mathcal{H} = \langle h_1, \dots, h_n \rangle$ be a tuple of admissible heuristics for regular fixed-cost transition system $\langle \mathcal{T}, cost \rangle$, and let $\langle w_1, \dots, w_n \rangle$ be a solution to the linear program that maximizes $\sum_{i=1}^n (w_i \cdot h_i(s, cost))$ subject to

$$\sum_{i \in \{1, \dots, n\} : l \in \mathcal{A}(h_i)} w_i \leq 1 \text{ for all } l \in \mathcal{L}(\mathcal{T})$$

$$w_i \geq 0 \text{ for all } 1 \leq i \leq n.$$

Then, the post-hoc optimization cost partitioning is the tuple $\mathcal{C} = \langle w_1 \cdot cost_1, \dots, w_n \cdot cost_n \rangle$, where $cost_i(l) = cost(l)$ if $l \in \mathcal{A}(h_i)$ and $cost_i(l) = 0$ otherwise. We write h^{PhO} for the heuristic that is cost-partitioned with the post-hoc optimization cost partitioning.

Example We illustrate post-hoc optimization with the two example abstraction heuristics h_1 and h_2 in Figure 3.1. For state s_1 the post-hoc opti-

mization LP maximizes the sum $w_1 \cdot h_1(s_1, cost) + w_2 \cdot h_2(s_1, cost) = w_1 \cdot 5 + w_2 \cdot 5 = 5 \cdot (w_1 + w_2)$ subject to the constraints

$$w_1 + w_2 \leq 1 \text{ (for } o_1) \quad (3.4)$$

$$w_2 \leq 1 \text{ (for } o_2) \quad (3.5)$$

$$w_1 + w_2 \leq 1 \text{ (for } o_3) \quad (3.6)$$

$$w_1 \leq 1 \text{ (for } o_4) \quad (3.7)$$

$$w_1 \geq 0 \quad (3.8)$$

$$w_2 \geq 0. \quad (3.9)$$

Since we can ignore constant factors in the optimization function of an LP, the objective is to maximize the sum $w_1 + w_2$. The constraints 3.4, 3.8 and 3.9 imply all other constraints, so any solution that satisfies $w_1 + w_2 = 1$, $w_1 \geq 0$ and $w_2 \geq 0$ is optimal, e.g., $\langle w_1 = 1, w_2 = 0 \rangle$, $\langle w_1 = 0, w_2 = 1 \rangle$ or $\langle w_1 = 0.25, w_2 = 0.75 \rangle$. The heuristic value for state s_1 is the objective value of the LP, i.e., $h^{\text{PhO}}(s_1, cost) = 5$.

Post-hoc optimization always generates a non-negative cost partitioning, i.e., one where all component costs are non-negative. Following the results of Pommerening et al. (2015) on general cost partitioning, one might wonder if h^{PhO} could be strengthened by dropping the non-negativity constraint $w_i \geq 0$. However, this does not work as expected, as $\sum_{i=1}^n (w_i \cdot h_i(s, cost))$ is no longer an admissible estimate without this constraint. The reason for this is that negative weighting changes which paths are optimal in a state space.

3.4 Zero-One Cost Partitioning

In a zero-one cost partitioning (Edelkamp 2006; Haslum et al. 2005), the whole cost of each label is assigned to (at most) a single component heuristic.

Definition 3.4 Zero-one cost partitioning.

Given a regular fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ and a tuple of admissible heuristics $\mathcal{H} = \langle h_1, \dots, h_n \rangle$, a tuple $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$ is a zero-one cost partitioning if for each $l \in \mathcal{L}$ we have $cost_i(l) = cost(l)$ for at most one $cost_i \in \mathcal{C}$ and $cost_j(l) = 0$ for all other $cost_j \in \mathcal{C}$.

For a state space with l labels and n admissible heuristics, Definition 3.4 allows for $(n + 1)^l$ different zero-one cost partitionings. The question is therefore how to obtain an informative zero-one cost partitioning. The only method considered in the literature is a greedy algorithm that iterates over the heuristics in a specified order and assigns the cost of each label to the first heuristic that is affected by this label. Due to this greedy assign-

ment, the algorithm is susceptible to the order in which the heuristics are considered.

Definition 3.5 Heuristic Orders.

Let \mathcal{H} be a finite set of heuristics. An order ω of \mathcal{H} is a tuple $\langle h_1, \dots, h_n \rangle$ that contains each heuristic in \mathcal{H} exactly once. We write $\Omega(\mathcal{H})$ for the set of all orders of \mathcal{H} .

Definition 3.6 Greedy zero-one cost partitioning.

Let $\langle \mathcal{T}, cost \rangle$ be a regular fixed-cost transition system and let \mathcal{H} be a set of admissible heuristics for \mathcal{T} .

For a given order $\omega = \langle h_1, \dots, h_n \rangle \in \Omega(\mathcal{H})$, the greedy zero-one cost partitioning is the tuple $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$, where

$$cost_i(l) = \begin{cases} cost(l) & \text{if } l \in \mathcal{A}(h_i) \text{ and } l \notin \bigcup_{j=1}^{i-1} \mathcal{A}(h_j) \\ 0 & \text{otherwise} \end{cases}$$

for all $l \in \mathcal{L}$. We write h_ω^{GZOC} for the heuristic that is cost-partitioned by greedy zero-one cost partitioning for order ω .

Each greedy zero-one cost partitioning is a zero-one cost partitioning as the cost of each label is assigned to at most one heuristic (the first one in the order affected by the label).

Example Consider again the two abstraction heuristics h_1 and h_2 and the cost function $cost$ from Figure 3.1. Depending on the heuristic order we can obtain two different greedy zero-one cost partitionings. The order $\langle h_1, h_2 \rangle$ yields the cost partitioning $\langle \langle 4, 0, 4, 1 \rangle, \langle 0, 1, 0, 0 \rangle \rangle$ and consequently $h_{\langle h_1, h_2 \rangle}^{GZOC}(s_1, cost) = 5$. Similarly, using the order $\langle h_2, h_1 \rangle$ leads to the cost partitioning $\langle \langle 4, 1, 4, 0 \rangle, \langle 0, 0, 0, 1 \rangle \rangle$, which also results in $h_{\langle h_2, h_1 \rangle}^{GZOC}(s_1, cost) = 5$. While the two orders yield the same heuristic value for s_1 , we have $h_{\langle h_1, h_2 \rangle}^{GZOC}(s_3, cost) = 1$ and $h_{\langle h_2, h_1 \rangle}^{GZOC}(s_3, cost) = 4$.

3.5 Uniform Cost Partitioning

Katz and Domshlak (2008) proposed uniform cost partitioning, where the cost of each label is distributed uniformly among all heuristics affected by this label.

Definition 3.7 Uniform cost partitioning.

Given a regular fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ and a tuple of admissible heuristics $\mathcal{H} = \langle h_1, \dots, h_n \rangle$ for \mathcal{T} , the uniform cost partitioning is the tuple $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$, where for all $l \in \mathcal{L}$

$$cost_i(l) = \begin{cases} \frac{cost(l)}{|\{h \in \mathcal{H} \mid l \in \mathcal{A}(h)\}|} & \text{if } l \in \mathcal{A}(h_i) \\ 0 & \text{otherwise.} \end{cases}$$

We write h^{UCP} for the heuristic that is cost-partitioned by uniform cost partitioning.

Unlike (greedy) zero-one cost partitioning, uniform cost partitioning is not affected by the order in which the heuristics are considered.

Example In our example from Figure 3.1 the original cost function is $cost = \langle 4, 1, 4, 1 \rangle$. For the two abstraction heuristics h_1 and h_2 the uniform cost partitioning is $\langle cost_1, cost_2 \rangle$ with $cost_1 = \langle 2, 0, 2, 1 \rangle$ and $cost_2 = \langle 2, 1, 2, 0 \rangle$. The resulting heuristic value for s_1 is $h^{\text{UCP}}(s_1, cost) = 3 + 3 = 6$.

3.6 Canonical Heuristic

Haslum et al. (2007) introduced the canonical heuristic as a heuristic that allows the combination of information from multiple pattern database heuristics. We give a definition for general admissible heuristics.

Definition 3.8 Canonical Heuristic.

Let \mathcal{H} be a set of admissible heuristics for regular fixed-cost transition system $\langle \mathcal{T}, cost \rangle$, and let $MIS(\mathcal{H})$ be the set of all maximal (w.r.t. set inclusion) subsets of \mathcal{H} such that all heuristics in each subset are pairwise independent. The canonical heuristic in state $s \in S(\mathcal{T})$ is

$$h^{\text{CAN}}(s, cost) = \max_{\sigma \in MIS(\mathcal{H})} \sum_{h \in \sigma} h(s, cost).$$

The canonical heuristic implicitly computes the maximum over *multiple* cost-partitioned heuristics: for each maximal independent subset $\langle h_1, \dots, h_n \rangle \in MIS(\mathcal{H})$ we can compute a cost partitioning $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$ with $cost_i(l) = cost(l)$ if l affects h_i and $cost_i(l) = 0$ otherwise. Since by definition all heuristics are pairwise independent, \mathcal{C} only uses the cost of each label in at most one cost function. Therefore, \mathcal{C} is a zero-one cost partitioning. We investigate the relationships between the canonical heuristic and other cost partitioning algorithms in more detail in Chapter 10.

Example The two abstraction heuristics h_1 and h_2 from Figure 3.1 are not independent since operators o_1 and o_3 affect both heuristics. Therefore, the canonical heuristic is the maximum over the individual heuristics. For example,

$$\begin{aligned} h^{\text{CAN}}(s_1, cost) &= \max(h_1(s_1, cost), h_2(s_1, cost)) = \max(5, 5) = 5 \text{ and} \\ h^{\text{CAN}}(s_3, cost) &= \max(h_1(s_3, cost), h_2(s_3, cost)) = \max(1, 4) = 4. \end{aligned}$$

Computing the canonical heuristic involves finding maximal independent subsets of a given set of component heuristics. This is equivalent to

finding maximal cliques in an undirected graph that has an edge between two heuristics if no label affects both of them. Since computing even a single maximal clique is NP-hard (Garey and Johnson 1979), computing the canonical heuristic is NP-hard as well. Pommerening et al. (2013) showed that post-hoc optimization dominates the canonical heuristic even though h^{PhO} is polynomial. The underlying reason for this difference is that finding maximal cliques corresponds to solving an integer program. While linear programs can be solved in polynomial time, solving integer programs is NP-hard (Karp 1972).

Part I

CARTESIAN ABSTRACTIONS

Counterexample-guided abstraction refinement (CEGAR) is a method for incrementally computing abstractions of transition systems. We propose a CEGAR algorithm for computing abstraction heuristics for optimal classical planning. Starting from a coarse abstraction of the planning task, we iteratively compute an optimal abstract solution, check if and why it fails for the concrete planning task and refine the abstraction so that the same failure cannot occur in future iterations. A key ingredient of our approach is a novel class of abstractions for classical planning tasks that admits efficient and very fine-grained refinement. Since a single abstraction usually cannot capture enough details of the planning task, we also introduce two methods for producing diverse sets of heuristics within this framework, one based on goal atoms, the other based on landmarks. In order to sum their heuristic estimates admissibly we introduce a new cost partitioning algorithm called saturated cost partitioning. We show that the resulting heuristics outperform other state-of-the-art abstraction heuristics in many benchmark domains.

CARTESIAN ABSTRACTION REFINEMENT

Counterexample-guided abstraction refinement is an established technique for model checking in large systems (Clarke et al. 2003). The idea is to start from a coarse (i.e., small and inaccurate) abstraction, then iteratively improve (refine) the abstraction in only the necessary places. In model checking, this means that we search for error traces (behaviors that violate the system property we want to verify) in the abstract system, test if these error traces generalize to the actual system (called the *concrete system*), and only if not, refine the abstraction in such a way that this particular error trace is no longer an error trace of the abstraction.

In model checking, CEGAR is usually used to prove the absence of an error trace. Here, we use CEGAR to derive heuristics for optimal state-space search, and hence our CEGAR procedure does not have to completely solve the problem: abstraction refinement can be interrupted at any time to derive an admissible search heuristic.

4.1 Cartesian Abstractions

We want to construct compact and informative abstractions through an iterative refinement process. Choosing a suitable class of abstractions is critical for this. For example, *pattern databases* (Edelkamp 2001) do not allow fine-grained refinement steps, as every refinement at least doubles the number of abstract states. *Merge-and-shrink* abstractions (Helmert et al. 2007, 2014) do not maintain efficiently usable representations of the preimage of an abstract state, which makes their refinement complicated and expensive.

Because of these and other shortcomings, we introduce a new class of abstractions for planning tasks that is particularly suitable for fine-grained abstraction refinement.

Definition 4.1 Cartesian sets and Cartesian abstractions.

A set of states for a planning task with variables $\langle v_1, \dots, v_n \rangle$ is called Cartesian if it is of the form $A_1 \times A_2 \times \dots \times A_n$, where $A_i \subseteq \text{dom}(v_i)$ for all $1 \leq i \leq n$.

An abstraction is called Cartesian if all its abstract states are Cartesian sets.

For an abstract state $a = A_1 \times \dots \times A_n$, we define $\text{dom}(v_i, a) = A_i \subseteq \text{dom}(v_i)$ for all $1 \leq i \leq n$ as the set of values that variable v_i can have in abstract state a .

Figure 4.1 shows an example Cartesian abstraction for the Gripper task. The Cartesian sets $\{A\} \times \{A, G\}$, $\{B\} \times \{A, G\}$ and $\{A, B\} \times \{B\}$ are the

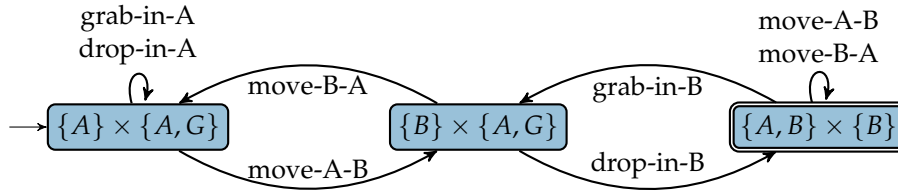


Figure 4.1: Example Cartesian abstraction of the Gripper example. In the left and center state we know where the robot is, but not whether its gripper holds the ball, whereas in the abstract goal state we ignore the position of the robot.

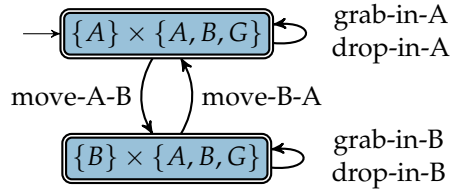
states in the abstract transition system. The heuristic h^\sim maps each state to the respective abstract goal distance (2, 1 or 0 — under the unit cost function).

The name “Cartesian abstraction” was coined in the model-checking literature by Ball et al. (2001) for a concept essentially equivalent to Definition 4.1. (Direct comparisons are difficult due to different state models.) We discuss their work in Section 4.5.

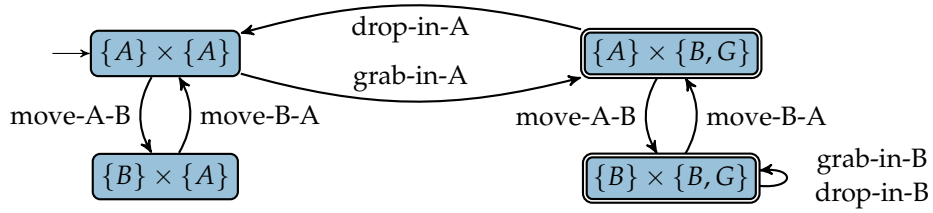
Cartesian abstractions form a fairly general class; e.g., they include projections (the abstractions underlying pattern databases) and domain abstractions (Hernádvölgyi and Holte 2000) as special cases. Unlike these, general Cartesian abstractions can have arbitrarily different levels of granularity in different parts of the abstract state space. One abstract state might correspond to a single concrete state while another abstract state corresponds to half of the states of the task.

We illustrate the relationships between different classes of abstractions with example abstractions of our Gripper task. Figure 4.2a shows the abstract transition system induced by the projection to the pattern $\{robot\}$. Clearly, this abstraction is also a domain abstraction, Cartesian abstraction and merge-and-shrink abstraction. When we additionally partition the domain for variable *ball* into the groups $\{A\}$ and $\{B, G\}$, the resulting abstraction is not a projection anymore (Figure 4.2b). It is however still a domain abstraction. A further split of state $\{B\} \times \{B, G\}$ into the two states $\{B\} \times \{B\}$ and $\{B\} \times \{G\}$ yields the Cartesian abstraction in Figure 4.2c. Since not all domains are split equally for all states, it is not a domain abstraction anymore. Combining the states $\{A\} \times \{B, G\}$ and $\{B\} \times \{A\}$ results in the transition system in Figure 4.2d. This abstraction is not Cartesian anymore, but can be expressed in the merge-and-shrink formalism.

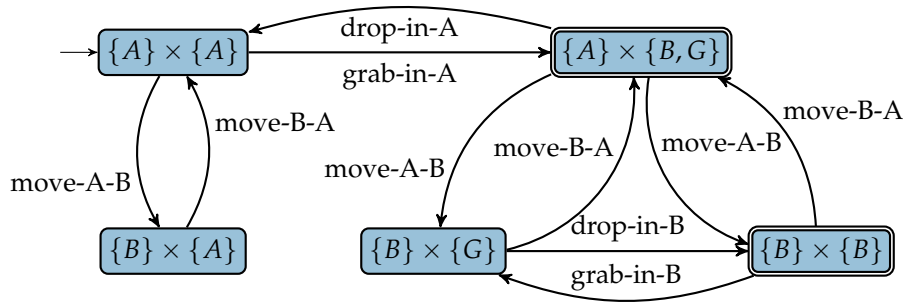
Merge-and-shrink abstractions are even more general than Cartesian abstractions because *every* abstraction function can be represented as a merge-and-shrink abstraction, but not necessarily compactly (Helmert et al. 2015). It is open whether every Cartesian abstraction has an equivalent



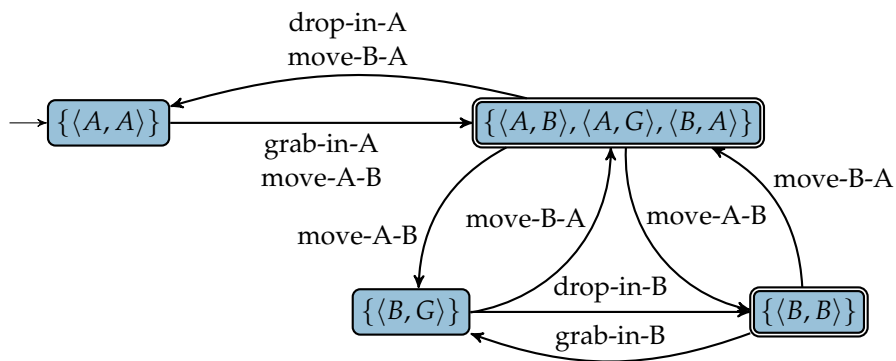
(a) Projection.



(b) Domain abstraction.



(c) Cartesian abstraction.



(d) Merge-and-shrink abstraction.

Figure 4.2: Example abstractions of the Gripper task for different classes of abstractions. The captions state the most specific class each abstraction belongs to.

merge-and-shrink abstraction whose representation is at most polynomially larger.

In the following theorem we collect some properties that make Cartesian sets interesting for CEGAR in planning. (We remind the reader that a *partition* of a set X is a collection of subsets $X_1, \dots, X_k \subseteq X$ that are pairwise disjoint and jointly exhaustive, i.e., $X_i \cap X_j = \emptyset$ for all $i \neq j$ and $\bigcup_{i=1}^k X_i = X$. Some definitions additionally require that all subsets are non-empty, but we do not need to make this restriction here.)

Theorem 4.1 Properties of Cartesian sets.

Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task.

- (P1) The set of goal states of Π is Cartesian.
- (P2) For all operators $o \in \mathcal{O}$, the set of states in which o is applicable is Cartesian.
- (P3) The intersection of Cartesian sets is Cartesian.
- (P4) For all operators $o \in \mathcal{O}$, the regression of a Cartesian set with respect to o is Cartesian.
- (P5) If $b \subseteq a$ and $c \subseteq a$ are disjoint Cartesian subsets of the Cartesian set a , then a can be partitioned into Cartesian sets d and e with $b \subseteq d$ and $c \subseteq e$.
- (P6) If $c \subseteq a$ is a Cartesian subset of the Cartesian set a and $s \in a \setminus c$, then a can be partitioned into Cartesian sets d and e with $s \in d$ and $c \subseteq e$.

Proof. Let $\mathcal{V} = \langle v_1, \dots, v_n \rangle$.

(P1) The set of goal states is $A_1 \times \dots \times A_n$ with

$$A_i = \begin{cases} \{s_\star[v_i]\} & \text{if } v_i \in \text{vars}(s_\star) \\ \text{dom}(v_i) & \text{otherwise} \end{cases}$$

(P2) The set of states where operator o is applicable is $A_1 \times \dots \times A_n$ with

$$A_i = \begin{cases} \{\text{pre}(o)[v_i]\} & \text{if } v_i \in \text{vars}(\text{pre}(o)) \\ \text{dom}(v_i) & \text{otherwise} \end{cases}$$

(P3) The intersection of two Cartesian sets $A_1 \times \dots \times A_n$ and $B_1 \times \dots \times B_n$ is $(A_1 \cap B_1) \times \dots \times (A_n \cap B_n)$.

(P4) The regression of Cartesian set $b = B_1 \times \dots \times B_n$ with respect to operator $o \in \mathcal{O}$ is $\text{regr}(b, o) = A_1 \times \dots \times A_n$ with

$$A_i = \begin{cases} B_i & \text{if } v_i \notin \text{vars}(\text{post}(o)) \\ \emptyset & \text{if } v_i \in \text{vars}(\text{post}(o)) \text{ and } \text{post}(o)[v_i] \notin B_i \\ \text{pre}(o)[v_i] & \text{if } v_i \in \text{vars}(\text{pre}(o)) \text{ and } \text{post}(o)[v_i] \in B_i \\ \text{dom}(v_i) & \text{if } v_i \in \text{vars}(\text{eff}(o)) \setminus \text{vars}(\text{pre}(o)) \text{ and } \text{post}(o)[v_i] \in B_i \end{cases}$$

To see that exactly one of the four cases applies in each situation, note that cases 2–4 all cover situations with $v_i \in \text{vars}(\text{post}(o))$ because $\text{vars}(\text{post}(o)) = \text{vars}(\text{pre}(o)) \cup \text{vars}(\text{eff}(o))$. The further distinctions in these cases are whether $\text{post}(o)[v_i] \in B_i$ (in cases 3–4, but not in case 2) and whether $v_i \in \text{vars}(\text{pre}(o))$ (in case 3, but not in case 4).

(P5) Let $a = A_1 \times \dots \times A_n$, $b = B_1 \times \dots \times B_n$ and $c = C_1 \times \dots \times C_n$. Set $X_i = B_i \cap C_i$ for all $1 \leq i \leq n$. Let j be an index such that $X_j = \emptyset$. Such an index must exist because otherwise we can select arbitrary values $x_i \in X_i$ for all $1 \leq i \leq n$ to obtain $\langle x_1, \dots, x_n \rangle \in b \cap c$, contradicting that b and c are disjoint.

Because $X_j = B_j \cap C_j = \emptyset$, we can partition A_j into D_j and E_j in such a way that $B_j \subseteq D_j$ and $C_j \subseteq E_j$, for example by setting $D_j = B_j$ and $E_j = A_j \setminus B_j$. Then $d = A_1 \times \dots \times A_{j-1} \times D_j \times A_{j+1} \times \dots \times A_n$ and $e = A_1 \times \dots \times A_{j-1} \times E_j \times A_{j+1} \times \dots \times A_n$ have the required property.

(P6) Follows from the previous property by setting $b = \{s\}$, which is a Cartesian set.

□

Next, we describe our abstraction refinement algorithm, provide an example, analyze its time complexity, and discuss some implementation details.

4.2 Abstraction Refinement Algorithm

We begin by describing the main loop of our algorithm before explaining the details of the underlying functions.

Main Loop. The main loop of the refinement algorithm is shown in Algorithm 1. At every time, the algorithm maintains an abstract fixed-cost transition system $\langle \mathcal{T}', \text{cost} \rangle$, where cost is the original cost function of the planning task. The transition system is represented as an explicit labeled digraph. Initially, \mathcal{T}' is the trivial abstract transition system, containing only one abstract state a_0 , which covers all concrete states of the planning task (line 2). Then the algorithm iteratively refines \mathcal{T}' until a termination criterion is satisfied (usually a time and/or memory limit, see line 3), the task is proven unsolvable (lines 5–6) or a concrete plan is found (lines 8–9).

Each iteration of the refinement loop first computes an optimal abstract goal trace τ' for the abstract initial state (line 4). If no such trace exists (τ' is “no trace”), the abstract task is unsolvable, and hence the concrete task is also unsolvable: we are done (lines 5–6).

Algorithm 1 Main loop. For a given planning task, returns a solution, proves that no solution exists, or returns an abstraction of the task (for example to be used as the basis of a heuristic function). All algorithms operate on the planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ with $\mathcal{V} = \langle v_1, \dots, v_n \rangle$ and cost function $cost$.

```

1: function CEGAR()
2:    $\langle \mathcal{T}', cost \rangle \leftarrow \text{TRIVIALABSTRACTION}()$ 
3:   while not TERMINATIONCONDITION() do
4:      $\tau' \leftarrow \text{FINDOPTIMALTRACE}(\langle \mathcal{T}', cost \rangle)$ 
5:     if  $\tau'$  is “no trace” then
6:       return task is unsolvable
7:      $\varphi \leftarrow \text{FINDFLAW}(\tau')$ 
8:     if  $\varphi$  is “no flaw” then
9:       return plan extracted from  $\tau'$ 
10:     $\mathcal{T}' \leftarrow \text{REFINE}(\mathcal{T}', \varphi)$ 
11:  return  $\mathcal{T}'$ 

```

Otherwise, we try to convert τ' into a concrete goal trace in the FINDFLAW function (line 7). If the conversion succeeds, i.e., τ' contains no flaw, we return the concrete plan extracted from τ' (lines 8–9). If the conversion fails, FINDFLAW returns the first encountered flaw φ , i.e., a reason for why the conversion failed. Afterwards, we refine \mathcal{T}' such that the same flaw φ cannot be encountered in future iterations (line 10).

In each step of the loop the goal distances of all states can only increase. Without time or memory limits the resulting abstraction heuristic monotonically increases in accuracy with each refinement, and we will eventually find an optimal concrete plan or prove that no concrete plan exists. If we hit a time or memory limit (line 3), we abort the loop and return the refined abstract transition system (line 11).

Trace Verification. The FINDFLAW function in Algorithm 2 attempts to convert the abstract goal trace $\tau' = \langle a_0 \xrightarrow{o_1} a_1, a_1 \xrightarrow{o_2} a_2, \dots, a_{k-1} \xrightarrow{o_k} a_k \rangle$ into a *concrete* goal trace $\langle s_0 \xrightarrow{o_1} s_1, s_1 \xrightarrow{o_2} s_2, \dots, s_{k-1} \xrightarrow{o_k} s_k \rangle$ starting from the initial state s_0 such that $[s_i] = a_i$ for all $0 \leq i \leq k$ and s_k is a goal state. The conversion procedure starts from the initial state $s = s_0$ of the concrete task (line 2) and iteratively applies the next operator o in τ' until one of the following flaws is encountered:

1. Operator o is not applicable in concrete state s (line 4).
2. For abstract and concrete transitions $a \xrightarrow{o} b$ and $s \xrightarrow{o} s[[o]]$ we have that $[s] = a$ but $[s[[o]]] \neq b$: the concrete and abstract traces diverge (line 6). This can happen because abstract transition systems are not

Algorithm 2 Trace verification. Tries to convert a given abstract goal trace τ' into a concrete goal trace. If the conversion fails, returns the first encountered “flaw”, i.e., a pair of a concrete state s and a Cartesian set c with the following property: converting the trace failed because s is not part of c . If the conversion succeeds, the function returns “no flaw”.

```

1: function FINDFLAW( $\tau'$ )
2:    $s \leftarrow s_0$ 
3:   for each  $(a \xrightarrow{o} b) \in \tau'$  do
4:     if  $o$  is not applicable in  $s$  then
5:       return  $\langle s, [s] \cap (C_1 \times \dots \times C_n) \rangle$  with
           
$$C_i = \begin{cases} \{pre(o)[v_i]\} & \text{if } v_i \in vars(pre(o)) \\ dom(v_i) & \text{otherwise} \end{cases}$$

6:     if  $b$  does not include  $s[[o]]$  then
7:       return  $\langle s, [s] \cap regr(b, o) \rangle$ 
8:      $s \leftarrow s[[o]]$ 
9:   if  $s$  is not a goal state then
10:    return  $\langle s, [s] \cap (C_1 \times \dots \times C_n) \rangle$  with
           
$$C_i = \begin{cases} \{s_\star[v_i]\} & \text{if } v_i \in vars(s_\star) \\ dom(v_i) & \text{otherwise} \end{cases}$$

11:  return “no flaw”

```

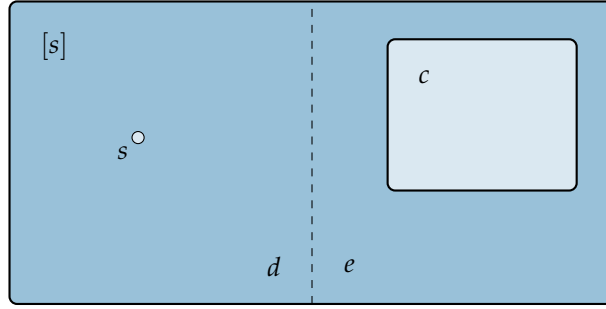


Figure 4.3: Illustration of how an abstract state $[s]$ must be split into two new abstract states d and e for a flaw $\langle s, c \rangle$.

necessarily deterministic: the same abstract state can have multiple outgoing transitions with the same label.

3. The concrete goal trace has been completed, but the last concrete state s is not a goal state (line 9).

In each case we can find a Cartesian set $c \subseteq [s]$ with the following property: converting the goal trace failed because s is not part of c . Depending on the case, c is

1. the set of concrete states in $[s]$ in which o is applicable (line 5),
2. the set of concrete states in $[s]$ from which we can reach b by applying o (line 7), or
3. the set of concrete goal states in $[s]$ (line 10).

Properties P1, P2, P3 and P4 from Theorem 4.1 entail that c is always a Cartesian set.

Refinement. After a flaw $\langle s, c \rangle$ has been identified, it serves as the basis for refining the abstraction. The goal here is to split $[s]$ into two abstract states in such a way that the same flaw cannot occur again after the refinement.

Property P6 from Theorem 4.1 guarantees that it is always possible to partition the abstract state $[s]$ into two Cartesian sets d and e that separate s from c (i.e., $s \in d$ and $c \subseteq e$). This is illustrated in Figure 4.3. For the respective cases this split ensures that

1. o is inapplicable in all concrete states in d ,
2. applying o in any state in d cannot lead to a state in $[s[o]]$, and
3. d contains no concrete goal states.

Note that to make progress in the abstraction refinement process, it is important that the partitioning of $[s]$ into d and e is proper, i.e., both subsets are proper subsets of $[s]$. This is equivalent to the requirement that both subsets are non-empty because a partition of a set into two subsets is non-proper iff one of the subsets is the whole set and the other one is empty. Hence, we now argue why $d \neq \emptyset$ and $e \neq \emptyset$.

The former is easy to see because $s \in d$. To see why $e \neq \emptyset$, we observe that $c \subseteq e$ and that in each of the three cases, c must be non-empty:

1. If c is the set of concrete states in $[s]$ in which o is applicable, c must contain a state in which o is applicable, since o induces a transition starting in $[s]$.
2. If c is the set of concrete states in $[s]$ from which we can reach b by applying o , c must contain a state s' in which applying o leads to $s'[[o]]$ with $[s'[[o]]] = b$, since o induces the transition $[s] \xrightarrow{o} b$.
3. If c is the set of concrete goal states in $[s]$, c cannot be empty because $[s]$ is an abstract goal state and \mathcal{T}' is an induced abstraction.

Because every refinement step for the flaw $\langle s, c \rangle$ separates s from c , it is easy to see that the same flaw can never be encountered in future iterations of the main loop. In every iteration, some abstract state is split into two smaller abstract states, and therefore the abstraction becomes increasingly more fine-grained. This implies that the main loop must eventually terminate, even without specifying a termination condition in line 3, either because no more abstract trace can be found or because the abstract trace corresponds to a concrete solution.

The `REFINE` function in Algorithm 3 shows the refinement process. It splits $[s]$ into two new abstract states d and e as explained above and updates the abstract transition system by replacing $[s]$ with d and e . In general, there can be many possible splits that separate s from c , and hence many possible choices of d and e . We discuss this choice in Section 4.3.

Next, the `REFINE` function “rewires” the transitions. Since only a single state is split into two new states, the rewiring procedure only needs to make local changes to the transition system. Concretely, it needs to decide for each incoming and outgoing transition of $[s]$, including self-loops, whether the transition needs to be rewired from/to d , from/to e , or both. This check is done by `CHECKTRANSITION` in Algorithm 4. The last step of the `REFINE` function is to update the abstract initial state and abstract goal states, if necessary. Due to the way we split $[s]$ into d and e , e can never be the abstract initial state and d can never be an abstract goal state.

Example CEGAR Abstraction Figure 4.4a shows the initial abstraction for our Gripper example task II. The empty abstract goal trace $\langle \rangle$ fails to

Algorithm 3 Refinement. Given an abstract transition system \mathcal{T}' and a flaw φ , refines \mathcal{T}' by splitting the abstract state $[s]$ into two new abstract states and returns the resulting abstract transition system \mathcal{T}'' .

```

1: function REFINED( $\mathcal{T}', \varphi$ )
2:    $\langle S', \mathcal{L}', T', [s_0], S'_* \rangle \leftarrow \mathcal{T}'$ 
3:    $\langle s, c \rangle \leftarrow \varphi$ 
4:    $\langle d, e \rangle \leftarrow \text{SPLIT}(s, c)$ 
5:    $S'' \leftarrow (S' \setminus \{[s]\}) \cup \{d, e\}$ 
6:    $T'' \leftarrow \text{REWIRETRANSITIONS}(T', [s], d, e)$ 
7:   if  $[s] = [s_0]$  then
8:      $a_0 \leftarrow d$ 
9:   else
10:     $a_0 \leftarrow [s_0]$ 
11:   if  $[s] \in S'_*$  then
12:      $S''_* \leftarrow (S'_* \setminus \{[s]\}) \cup \{e\}$ 
13:   else
14:      $S''_* \leftarrow S'_*$ 
15:   return  $\langle S'', \mathcal{L}', T'', a_0, S''_* \rangle$ 

```

Algorithm 4 Transition check. Returns true iff operator o induces a transition between abstract states a and b .

```

1: function CHECKTRANSITION( $a, o, b$ )
2:   for each  $v \in \mathcal{V}$  do
3:     if  $v \in \text{vars}(\text{pre}(o))$  and  $\text{pre}(o)[v] \notin \text{dom}(v, a)$  then
4:       return false
5:     if  $v \in \text{vars}(\text{post}(o))$  and  $\text{post}(o)[v] \notin \text{dom}(v, b)$  then
6:       return false
7:     if  $v \notin \text{vars}(\text{post}(o))$  and  $\text{dom}(v, a) \cap \text{dom}(v, b) = \emptyset$  then
8:       return false
9:   return true

```

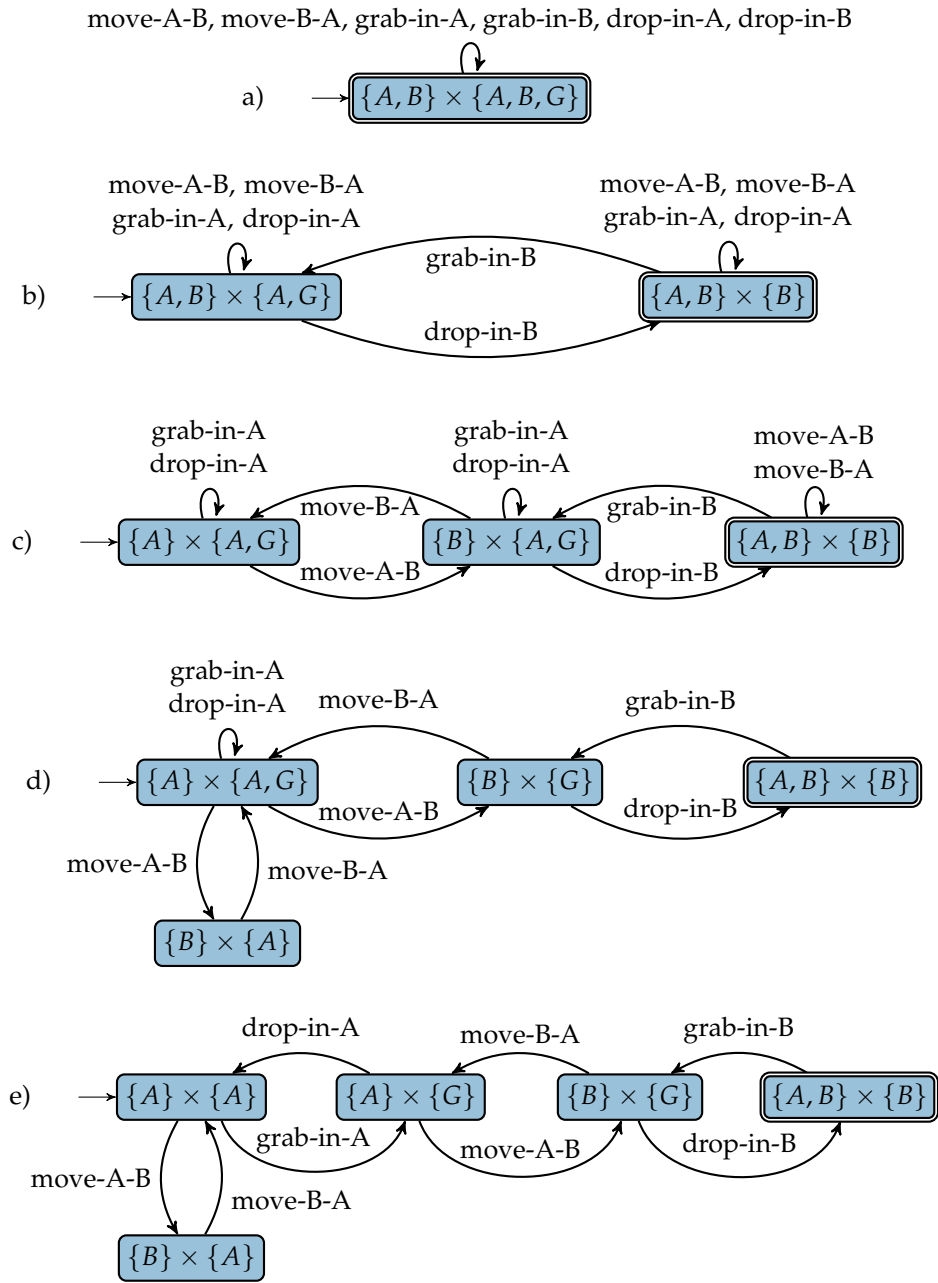


Figure 4.4: Refining the example abstraction.

solve Π because s_0 does not satisfy the goal. Therefore, **REFINE** splits $[s_0]$ based on the goal variable, leading to the finer abstraction in Figure 4.4b.

The abstract goal trace $\langle (\{A, B\} \times \{A, G\}) \xrightarrow{\text{drop-in-B}} (\{A, B\} \times \{B\}) \rangle$ in Figure 4.4b does not solve Π because two preconditions of **drop-in-B** are violated in s_0 : $ball \mapsto G$ and $robot \mapsto B$. We assume that **REFINE** performs a split based on variable $robot$ (a split based on $ball$ is also possible), which leads to Figure 4.4c.

A further refinement step, splitting on $ball$, yields the system in Figure 4.4d with the abstract goal trace $\langle (\{A\} \times \{A, G\}) \xrightarrow{\text{move-A-B}} (\{B\} \times \{G\}), (\{B\} \times \{G\}) \xrightarrow{\text{drop-in-B}} (\{A, B\} \times \{B\}) \rangle$. The first operator is applicable in s_0 and takes us into state s_1 with $s_1(robot) = B$ and $s_1(ball) = A$, but the second abstract state $a_1 = \{B\} \times \{G\}$ of the goal trace does not abstract s_1 : the abstract and concrete paths diverge. Regression from a_1 with respect to **move-A-B** yields the Cartesian set state $c = \{A\} \times \{G\}$, and hence **REFINE** must split the abstract initial state $[s_0]$ into two new states d and e in such a way that $s_0 \in d$ and $c \subseteq e$. The result of this refinement is shown in Figure 4.4e.

The plan for this abstraction is also a valid concrete plan, so we stop refining.

4.3 Implementation

In the following, we describe some of our implementation decisions.

Representation of Cartesian Sets. Even though we write $a = A_1 \times \dots \times A_n$ to denote a Cartesian set for the variable sequence $\langle v_1, \dots, v_n \rangle$, it is important to note that we do not store Cartesian sets as sets of concrete states, which would require exponential space. Instead, we only store the abstract domains $dom(v_i, a)$ for $1 \leq i \leq n$, requiring space linear in the number of atoms.

Refinement Strategy. As noted above, when handling the flaw $\langle s, c \rangle$, we have to partition $[s]$ into two new abstract states d and e with $s \in d$ and $c \subseteq e$. Due to the nature of Cartesian abstractions, the only way of splitting $[s]$ into two Cartesian sets d and e is to choose a single variable v with $s[v] \notin dom(v, c)$ and partition $dom(v, [s])$ into $dom(v, d)$ and $dom(v, e)$ (cf. the proof of property P6 in Theorem 4.1). All other abstract domains must remain the same as in $[s]$, i.e., $dom(v', d) = dom(v', e) = dom(v', [s])$ for all $v' \in \mathcal{V} \setminus \{v\}$. The question is how to choose v and how to partition $dom(v, [s])$.

Let us first consider the question of selecting a variable v on which to split $[s]$. After some preliminary experiments we chose the following

Algorithm 5 Abstract state lookup function. Given the concrete state s , return the abstract state $[s]$ by traversing the refinement hierarchy.

```

1: function GETABSTRACTSTATE( $s$ )
2:    $a \leftarrow$  root of the refinement hierarchy
3:   while HASCHILDREN( $a$ ) do
4:      $b, c \leftarrow$  GETCHILDREN( $a$ )
5:      $v \leftarrow$  GETSPLITVARIABLE( $a$ )
6:     if  $s[v] \in \text{dom}(v, b)$  then
7:        $a \leftarrow b$ 
8:     else
9:        $a \leftarrow c$ 
10:  return  $a$ 

```

variable-selection strategy, which we call “max-refined”. Out of the candidate variables, it selects the one that has already been refined the most in $[s]$, i.e., the variable v that minimizes the fraction $\frac{|\text{dom}(v, [s])|}{|\text{dom}(v)|}$ among all variables for which splits are feasible. We break ties by choosing the variable with the smallest index. This strategy clearly outperforms picking splits randomly and its inverse strategy, i.e., “min-refined”.

After selecting a variable v for which a split is feasible, we need to separate $s[v]$ from $\text{dom}(v, c)$. We must put $s[v]$ into d and $\text{dom}(v, c)$ into e . We are free in deciding where to put the remaining values $\text{dom}(v, [s]) \setminus (\{s[v]\} \cup \text{dom}(v, c))$. All other things being equal, we might expect that the abstract goal distance of d (the state we actually reached) might be higher than the one of e (the new state we would have wanted to reach), because the remainder of the abstract trace might correspond to a concrete solution from e , but definitely not from d . Therefore, we choose to put the remaining values into d , which might result in a greater increase of the average heuristic value.

Lookup Function. For the heuristic to be efficiently computable, we must be able to retrieve heuristic values very fast. The most critical operation here is computing the abstract state $[s]$ given a concrete state s . To make this operation efficient, we store a *refinement hierarchy* of split abstract states. This hierarchy is a binary tree of abstract states whose root is the first abstract state that was split, i.e., the only abstract state in the trivial abstract transition system. Whenever a state is split, the two resulting states become its child nodes. The leaves of the refinement hierarchy are the states in the final abstraction. In addition to the child nodes we also store the variable on which each abstract state was split. Figure 4.5 shows an example refinement hierarchy and GETABSTRACTSTATE in Algorithm 5

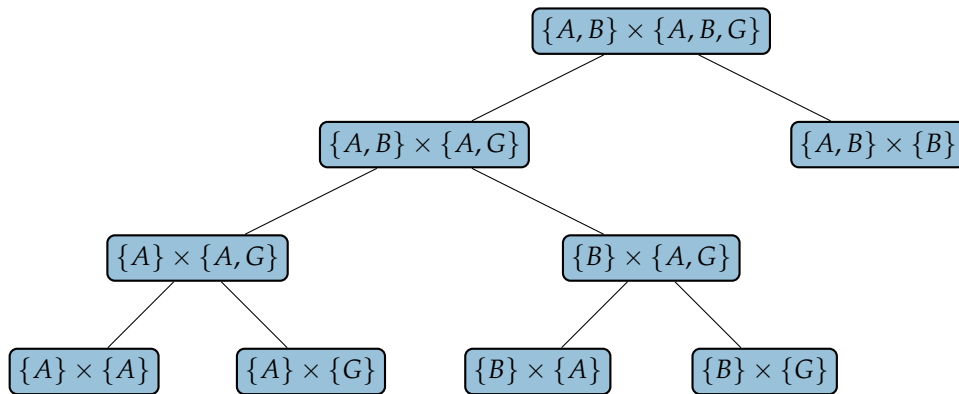


Figure 4.5: Refinement hierarchy for the final abstraction of the Gripper example from Figure 4.4e.

demonstrates how we use refinement hierarchies for looking up abstract states. We analyze the runtime complexity of this operation below.

A* Search. In principle, we could use any optimal algorithm for finding optimal traces in line 4 of Algorithm 1, such as Dijkstra’s algorithm. However, the `FINDOPTIMALTRACE` function runs significantly faster if we use A* with the following heuristic: every time we find an abstract trace τ , we update the goal distances of all states visited by τ . These heuristic values are admissible since τ is optimal. During each refinement we use the goal distance of the split state for the two new states. The heuristic remains admissible since refinements can only increase goal distances.

Self-loops. We store self-loops separately from state-changing transitions. This reduces both the time and the memory needed to refine an abstraction. The reason for the speedup is that the representation allows us to avoid the overhead of following self-loops in the `FINDOPTIMALTRACE` function. We save memory since instead of storing an operator and a destination state, as for state-changing transitions, we only need to store an operator for self-loops.

Informed Transition Check. Since we have to rewire numerous transitions during each refinement, we need to make this operation as fast as possible. Algorithm 4 shows a straightforward but inefficient implementation. The procedure decides whether a given hypothetical transition is part of the abstract transition system by checking for each variable whether the operator can change the value of the variable from one state to the other. While splitting a state on a variable v however, we can exploit that we only need to take v into account to determine whether the transition is valid. This greatly speeds up the refinement loop. In our implementation

we split the generic transition check into three specialized procedures, one for each type of transition: `REWIREINCOMINGTRANSITION` (Algorithm 10), `REWIREOUTGOINGTRANSITION` (Algorithm 11) and `REWIRESELFLOOP` (Algorithm 12). They are shown in Appendix A.1.

4.4 Theoretical Runtime Analysis

For our abstraction refinement algorithm to be useful, it has to be able to make refinements very fast. The following runtime guarantees show that the critical operations have adequate worst-case runtime complexities.

Theorem 4.2 Runtimes of operations on Cartesian sets.

If k is the number of atoms in a planning task Π , the following functions are computable in time $O(k)$ for Cartesian abstractions \sim :

- (R1)** *Compute the intersection of two Cartesian sets.*
- (R2)** *Compute the regression of operator o over a Cartesian set.*
- (R3)** *Given $s \in S(\Pi)$, compute $[s]_{\sim}$ and $h^{\sim}(s)$ (after abstract goal distances have been precomputed).*
- (R4)** *Given Cartesian sets a and b and operator o , decide if $a \xrightarrow{o} b$ is an abstract transition.*
- (R5)** *Given state $s \in S(\Pi)$ and Cartesian set a , decide whether a contains s .*

Proof. We represent Cartesian sets a as bit vectors where exactly the bits corresponding to atoms included in a are set. Let $\mathcal{V} = \langle v_1, \dots, v_n \rangle$ be the sequence of variables in Π .

- (R1)** Intersecting two Cartesian sets a and b with $a = A_1 \times \dots \times A_n$ and $b = B_1 \times \dots \times B_n$ by intersecting the bit vectors takes time $O(k)$.
- (R2)** The proof of property P4 of Theorem 4.1 on page 28 shows how the regression is computed. All membership tests can be computed in $O(1)$. Therefore, the worst-case time complexity is $O(k)$.
- (R3)** Algorithm 5 on page 37 shows how we compute the abstract state corresponding to a given concrete state. Its runtime is determined by the number of iterations and the time for each iteration. There can be at most k iterations, since in the worst case we split off one of the k atoms from the same abstract state k times. Because the set membership test in line 6 runs in constant time, computing $[s]_{\sim}$ given $s \in S(\Pi)$ runs in time $O(k)$. Once we have computed the abstract state, retrieving the heuristic value is a constant-time lookup operation. Therefore, also computing $h^{\sim}(s)$ given $s \in S(\Pi)$ takes time $O(k)$.

- (R4)** Algorithm 4 on page 34 shows the general procedure for checking if a transition exists between two states. The runtime of the third case in the loop dominates the runtimes of the other two because it involves the intersection of domains instead of a simple membership test. One intersection for variable $v \in \mathcal{V}$ runs in time $O(|dom(v)|)$. In the worst case an intersection is performed for each variable $v \in \mathcal{V}$, and thus the whole algorithm has the asymptotic runtime $O(\sum_{i=1}^n |dom(v_i)|) = O(k)$.
- (R5)** Let $b = \{s[v_1]\} \times \dots \times \{s[v_n]\}$ be the Cartesian set that only contains s . Constructing b takes time $O(k)$ since we represent it as a bit vector of length k . The intersection $a \cap b$ is non-empty iff a contains s . Using property R1 we can see that the whole operation runs in time $O(k)$.

□

4.5 Related Work

Abstraction is an important technique for model-checking large systems (Clarke et al. 1999). Counterexample-guided abstraction refinement was developed in this context to let abstractions focus on the system parts that matter for proving correctness or finding errors (Clarke et al. 2003).

Ball et al. (2001) were the first to use Cartesian abstractions for model checking. They use Boolean abstraction (a.k.a. predicate abstraction) to obtain a coarser version of the program they want to verify. For n Boolean predicates, each abstract state can be represented by a bit vector of size n . Since computing the transitions between abstract states in Boolean abstractions is often too expensive, they compute a Cartesian abstraction on top of the Boolean abstraction. For a given set of bit vectors they compute the corresponding abstract state as the smallest Cartesian product that contains all bit vectors. In contrast to our work, the Cartesian sets in their abstraction can therefore overlap.

Similarly to our work, they iteratively refine the abstractions with CEGAR. The main difference to our work is that they use symbolic model-checking, the predominant approach in that field, whereas we represent Cartesian abstractions explicitly. Another difference to our work is that they use CEGAR until an error is found or the system is proven correct, whereas we can stop the refinement at any time and use the resulting abstraction as a heuristic for A^* search.

Smaus and Hoffmann (2009) have explored the idea of using CEGAR to derive informative heuristics for model checking, although not with a focus on optimality. They use CEGAR to iteratively refine predicate abstraction heuristics for directed model-checking of timed automata via greedy

best-first search. As we demonstrate for the optimal classical planning setting in Chapter 7, they show that computing separate abstractions for each error condition is often beneficial. Since they do not enforce admissibility, they can combine the heuristics by maximizing or summing over them. Their results also show that there is no advantage in refining multiple paths instead of a single path in each iteration of the refinement loop. It remains to be tested if the same holds in our setting.

Despite the similarity between model checking and planning, CEGAR has not been thoroughly explored by the planning community. The work that comes closest to ours in a planning setting uses CEGAR for stochastic perfect information games, a generalization of Markov decision processes (Chatterjee et al. 2005). Stochastic perfect information games model two adversarial players and an uncertain environment. The authors propose an algorithm that iteratively refines an abstraction of the game using CEGAR. In each step, the algorithm finds and checks a winning abstract strategy (strategy 1) for player 1 in the concrete game. If strategy 1 does not correspond to a concrete strategy, it is refuted by constructing a winning abstract strategy for player 2 (strategy 2). If strategy 2 does not work in the concrete game, the abstract game is refined so that strategy 2 is eliminated. If either of the abstract strategies works in the concrete game, the respective player wins.

The authors also propose a variant of their algorithm for deterministic transition systems. It first compiles the planning task to a Boolean formula and then iteratively refines an abstraction that takes more and more Boolean variables into account.

Unfortunately, the paper contains no experimental evaluation or indication that either algorithm variant has been implemented. Both variants are based on blind search, and we believe they are very unlikely to deliver competitive performance. Moreover, the paper has several critical technical errors which make the main contribution (Algorithm 1) unsound.

Haslum (2012) introduces an algorithm for finding lower bounds on the solution cost of a planning task by iteratively “derelaxing” its delete relaxation. Our approach is similar in spirit, but technically quite different from Haslum’s because it is based on homomorphic abstraction rather than delete relaxation. As a consequence, our method performs shortest-path computations in abstract state spaces represented as explicit graphs in order to find abstract solutions, while Haslum’s approach exploits structural properties of delete-free planning tasks. More concretely, his algorithm iteratively traces solutions for the (refined) relaxed task in the original task (similarly to our algorithm) and combines atom conjunctions into additional atoms whenever tracing fails.

Another difference to our work is that Haslum uses his algorithm to prove lower bounds, whereas we use the obtained abstractions to com-

pute admissible heuristics for A^* search. In principle, both methods can be used for both purposes, but Haslum’s algorithm suffers from the fact that the number of operators grows exponentially in the number of added conjunctions. This makes evaluating a heuristic based on Haslum’s algorithm in a large number of states infeasible (Keyder et al. 2012).

Keyder et al. (2012) address this problem by proposing a new refinement scheme for delete relaxations which uses conditional effects to keep the number of operators linear in the number of added conjunctions. They iteratively refine the relaxed task until a given resource limit is hit. Afterwards, they compute a satisficing solution for the relaxed task with the FF heuristic (Hoffmann and Nebel 2001) and use it as a heuristic in greedy best-first search.

The same authors showed in a later paper (Keyder et al. 2014) that both Haslum’s original compilation method and their compilation method using conditional effects greatly benefit from ignoring operators that violate *mutexes*. In fact, their evaluation shows that Haslum’s approach together with mutex pruning is the compilation method of choice when refining relaxed tasks for LM-Cut (Helmert and Domshlak 2009).

Fickert and Hoffmann (2017) use Keyder et al.’s refinement algorithm to refine relaxed tasks online during search. Whenever they encounter local minima of the resulting heuristic, they learn new atom conjunctions and use them to improve the heuristic. We briefly tried online abstraction refinement with CEGAR in preliminary experiments, but the results were rather discouraging. It would be interesting to see whether more principled experiments in a larger design space achieve better results for online CEGAR using Cartesian abstractions.

SATURATED COST PARTITIONING

Even with the algorithmic improvements we discussed in Section 4.4, building a single big abstraction suffers from the problem of diminishing returns. This phenomenon is quantified by *Korf's conjecture*, which implies that in a unit-cost setting the maximum heuristic value in an abstract transition system grows (only) logarithmically in the number of abstract states (Korf 1997). For example, if the base of this logarithm is 2, each successive improvement of the heuristic value of the initial state by 1 might require doubling the number of abstract states: a prototypical example of diminishing returns. While Korf's conjecture makes many simplifying assumptions that do not generally hold, experiments have confirmed many times and for many different classes of abstractions that such diminishing returns almost always occur. (See Holte, 2013, for a detailed discussion of Korf's conjecture and its consequences.)

Intuitively, using only a single abstraction of a given task is often not enough to cover sufficiently many aspects of the task with a reasonable number of abstract states. Therefore, it is often beneficial to build multiple abstractions that focus on different aspects of the problem (Holte et al. 2006). This raises two questions: how do we come up with different abstractions, and how do we combine their heuristic estimates admissibly? To answer the second question, we introduce a new cost partitioning algorithm, which we describe next. In Chapter 6, we discuss ways to calculate diverse sets of abstractions.

All cost partitioning algorithms presented in Chapter 3, except for greedy zero-one cost partitioning, have in common that they need to keep all transition systems (or at least the information which labels affect which heuristics) in memory while computing the cost partitioning. Since we prefer a method that allows having only one transition system in memory at any time, greedy zero-one cost partitioning seems like a good fit. However, this algorithm suffers from a significant drawback: greedy zero-one cost partitioning considers the heuristics sequentially and gives the full costs of each label l to the first heuristic h affected by l , even if h does not need all of the costs to justify its heuristic estimates. In this case, we can obtain the same heuristic values even if we use lower costs. The unneeded costs can then be saved for subsequent heuristics.

For an example of this situation, consider the abstract transition system \mathcal{T}_h associated with abstraction heuristic h in Figure 5.1. The abstract states in \mathcal{T}_h are labeled with their goal distances ($h = X$). The cost function *cost*

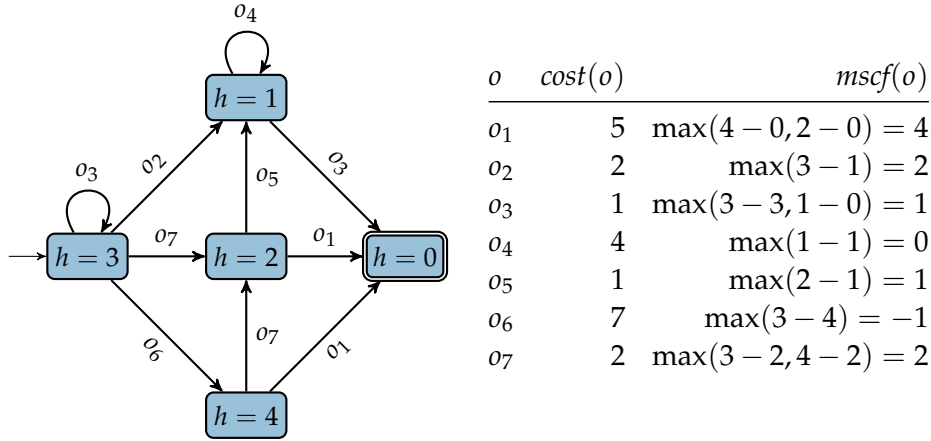


Figure 5.1: Left: abstract transition system of an example planning task. Every transition is labeled with an operator. Right: offered costs and minimum saturated costs that suffice to preserve all goal distances in the abstract transition system.

assigns o_1 the cost 5. Since o_1 affects h , greedy zero-one cost partitioning gives the complete cost of o_1 to h . However, if we reduce the cost of o_1 to 4, no goal distance changes. Consequently, we can “save” part of the cost for o_1 and use it for a different heuristic. A similar argument holds for operator o_4 . Since operator o_6 with $cost(o_6) = 7$ takes us one cost unit further away from the abstract goal state, we can assign it the cost -1 . In a subsequent heuristic, o_6 can then have a cost of $7 + 1 = 8$.

The observation that we can often use a lower cost function for a heuristic without changing any heuristic values, forms the basis of our new cost partitioning algorithm, which we call *saturated cost partitioning*. Its most important ingredient are *saturated cost functions*.

5.1 Saturated Cost Function

For a given heuristic h and cost function $cost$, we call a cost function *saturated* if it preserves all heuristic estimates that h yields under $cost$ and assigns each label l at most cost $cost(l)$.

Definition 5.1 Saturated Cost Function.

Let \mathcal{T} be a parameterized-cost transition system. Let h be a heuristic for \mathcal{T} and $cost \in \mathcal{C}(\mathcal{T})$ be a cost function for \mathcal{T} . A cost function scf is saturated for h and $cost$ if

1. $scf(l) \leq cost(l)$ for all labels $l \in \mathcal{L}(\mathcal{T})$ and
2. $h(s, scf) = h(s, cost)$ for all states $s \in S(\mathcal{T})$.

Note that we do not require that saturated cost functions are minimal since for some classes of heuristics (e.g., the h^{\max} heuristic by Bonet and Geffner, 2001) it is not guaranteed that there is a unique minimum. Using Definition 5.1, a saturated cost function exists for all heuristics h and cost functions $cost$, since $cost$ itself is a saturated cost function for h and $cost$. We call $cost$ a *trivial* saturated cost function in this case.

Obviously, only saturated cost functions scf with $scf(l) < cost(l)$ for at least one label l are useful for partitioning a cost function $cost$. If $scf = cost$, no costs are left for other heuristics. Whether and how we can compute a non-trivial saturated cost function for a given heuristic and cost function depends on the type of heuristic. We call functions that transform cost functions into saturated cost functions *saturators*.

Definition 5.2 Saturators.

Let \mathcal{T} be a parameterized-cost transition system and let h be a heuristic for \mathcal{T} . A function $saturate : \mathcal{C}(\mathcal{T}) \rightarrow \mathcal{C}(\mathcal{T})$ is a saturator for h if $saturate(cost)$ is a saturated cost function for heuristic h and all cost functions $cost \in \mathcal{C}(\mathcal{T})$.

It follows that there is at least one trivial saturator for all heuristics: the identity function that yields $saturate(cost) = cost$ for all cost functions $cost$. Before we introduce a minimum saturator for abstraction heuristics, we now explain the *saturated cost partitioning* algorithm, which works for any saturator.

5.2 Saturated Cost Partitioning Algorithm

Given a set of admissible heuristics \mathcal{H} , an order ω of \mathcal{H} and a *remaining cost function* $cost$, saturated cost partitioning iteratively considers the next heuristic h in ω , uses a saturator for h to compute a saturated cost function for h and $cost$ and subtracts the saturated costs from the remaining costs before turning to the next heuristic.

Definition 5.3 Saturated Cost Partitioning.

Let $\langle \mathcal{T}, cost \rangle$ be a regular fixed-cost transition system, let \mathcal{H} be a set of admissible heuristics for \mathcal{T} , let $\langle h_1, \dots, h_n \rangle \in \Omega(\mathcal{H})$ be an order of \mathcal{H} and let $\langle saturate_1, \dots, saturate_n \rangle$ be corresponding saturators. The saturated cost partitioning $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$ and the remaining cost functions $\langle remain_0, \dots, remain_n \rangle$ are defined by

$$\begin{aligned} remain_0 &= cost \\ cost_i &= saturate_i(remain_{i-1}) \\ remain_i &= remain_{i-1} - cost_i \end{aligned}$$

We write h_ω^{SCP} for the heuristic that is cost-partitioned by saturated cost partitioning for order ω .

Saturated cost functions may assign $-\infty$ to labels l that are never part of any optimal plan and can therefore have arbitrarily low costs without changing any heuristic estimates. As a consequence, we can assign arbitrarily high costs to l in all subsequent heuristics without affecting admissibility. Since ∞ is not included in the codomain of cost functions (and changing their definition would complicate other definitions and proofs), we use a large positive (finite) value M instead and define $remain_i(l) = M$ for all labels l with $cost_i(l) = -\infty$.

The quality of the resulting saturated cost partitioning strongly depends on the choice of saturators. For example, if we only use the identity function, all costs are assigned to the first heuristic, leaving no costs for subsequent heuristics. Ideally, we want saturators which return minimal saturated cost functions. It is open which properties a heuristic must have for it to possess a saturator that always returns minimum saturated cost functions.

5.3 Minimum Saturated Cost Function for Abstraction Heuristics

For explicitly represented abstraction heuristics there is always a unique minimum saturated cost function and it can be efficiently computed as shown in the following theorem.

Theorem 5.1 Minimum Saturated Cost Function for Abstraction Heuristics.

Let \mathcal{T} be a parameterized-cost concrete transition system, let h be an abstraction heuristic for \mathcal{T} with associated abstract transition system \mathcal{T}_h and let $cost \in \mathcal{C}(\mathcal{T}_h)$ be a cost function for \mathcal{T}_h . Furthermore, let $T_{fin}^h \subseteq T(\mathcal{T}_h)$ be the subset of transitions $a \xrightarrow{l} b \in T(\mathcal{T}_h)$ with $h_{\mathcal{T}_h}^*(b, cost) < \infty$. Then the minimum saturated cost function $mscf$ for h and $cost$ is defined as

$$mscf(l) = \begin{cases} -\infty & \text{if no transition in } T_{fin}^h \text{ has label } l \\ \max_{a \xrightarrow{l} b \in T_{fin}^h} (h_{\mathcal{T}_h}^*(a, cost) - h_{\mathcal{T}_h}^*(b, cost)) & \text{otherwise} \end{cases}$$

for all $l \in \mathcal{L}(\mathcal{T})$.

Proof. We first observe that $mscf(l)$ in the statement of the theorem is well-defined for every label $l \in \mathcal{L}(\mathcal{T})$. If no transition in T_{fin}^h has label l , the first case applies. Otherwise, the maximization in the second case is over a non-empty set, and $h_{\mathcal{T}_h}^*(a, cost) - h_{\mathcal{T}_h}^*(b, cost)$ for a given transition $a \xrightarrow{l} b$ is a difference of finite values and hence well-defined. (The value $h_{\mathcal{T}_h}^*(b, cost)$ is finite by definition of T_{fin}^h , and $h_{\mathcal{T}_h}^*(a, cost)$ is finite because a has a successor with finite heuristic value, namely b .)

We now show that $mscf$ satisfies the two properties of saturated cost functions from Definition 5.1:

- reg. 1. We need to show that $mscf(l) \leq cost(l)$ for all labels $l \in \mathcal{L}(\mathcal{T})$. In the case where $mscf(l) = -\infty$, this holds trivially, and otherwise there exists a transition $a \xrightarrow{l} b \in T(\mathcal{T}_h)$ with $mscf(l) = h_{\mathcal{T}_h}^*(a, cost) - h_{\mathcal{T}_h}^*(b, cost) \leq cost(l) + h_{\mathcal{T}_h}^*(b, cost) - h_{\mathcal{T}_h}^*(b, cost) = cost(l)$, where we use that $h_{\mathcal{T}_h}^*(a, cost) \leq cost(l) + h_{\mathcal{T}_h}^*(b, cost)$ by the triangle inequality.
- reg. 2. We must show $h(s, mscf) = h(s, cost)$ for all concrete states $s \in S(\mathcal{T})$.

From $mscf(l) \leq cost(l)$ for all $l \in \mathcal{L}(\mathcal{T})$ (see first part of this proof) and the fact that h is cost-monotonic (since it is an abstraction heuristic), we get that $h(s, mscf) \leq h(s, cost)$ for all states $s \in S(\mathcal{T})$.

It remains to show $h(s, mscf) \geq h(s, cost)$ for all concrete states $s \in S(\mathcal{T})$. Since the abstraction function is unaffected by changing the cost function, this is the case if $h_{\mathcal{T}_h}^*(a, mscf) \geq h_{\mathcal{T}_h}^*(a, cost)$ for all abstract states $a \in S(\mathcal{T}_h)$.

Let a_0 be any abstract state in $S(\mathcal{T}_h)$. If there is no goal trace for a_0 , we have $h_{\mathcal{T}_h}^*(a_0, mscf) = h_{\mathcal{T}_h}^*(a_0, cost) = \infty$. Otherwise, let $\tau = \langle a_0 \xrightarrow{l_1} a_1, \dots, a_{k-1} \xrightarrow{l_k} a_k \rangle$ be a goal trace for a_0 . All transitions in τ must be part of T_{fin}^h because clearly the goal is reachable from all states that τ traverses. We can bound the cost of τ under $mscf$ by

$$\begin{aligned}
\sum_{i=1}^k mscf(l_i) &\stackrel{(1)}{\geq} \sum_{i=1}^k (h_{\mathcal{T}_h}^*(a_{k-1}, cost) - h_{\mathcal{T}_h}^*(a_k, cost)) \\
&\stackrel{(2)}{=} \sum_{i=0}^{k-1} h_{\mathcal{T}_h}^*(a_k, cost) - \sum_{i=1}^k h_{\mathcal{T}_h}^*(a_k, cost) \\
&\stackrel{(3)}{=} h_{\mathcal{T}_h}^*(a_0, cost) - h_{\mathcal{T}_h}^*(a_k, cost) \\
&\stackrel{(4)}{=} h_{\mathcal{T}_h}^*(a_0, cost) - 0 \\
&= h_{\mathcal{T}_h}^*(a_0, cost),
\end{aligned}$$

where (1) uses that $mscf(l) \geq h_{\mathcal{T}_h}^*(a, cost) - h_{\mathcal{T}_h}^*(b, cost)$ for all transitions $a \xrightarrow{l} b \in T_{fin}^h$, (2) and (3) are basic arithmetic, and (4) uses that a_k is a goal state.

This shows that the cost of any plan for a_0 under $mscf$ is never lower than $h_{\mathcal{T}_h}^*(a_0, cost)$, the cost of an optimal plan under $cost$. This proves $h_{\mathcal{T}_h}^*(a, mscf) \geq h(a, cost)$ for all abstract states $a \in S(\mathcal{T}_h)$, concluding this part of the proof.

Finally, we show by contradiction that $mscf$ is minimal among all saturated cost functions for h and $cost$. Let $cost'$ be a cost function with $h(s, cost') = h(s, cost)$ for all concrete states $s \in S(\mathcal{T})$ and $cost'(l) < mscf(l)$ for some label $l \in \mathcal{L}(\mathcal{T})$. Since $cost'(l)$ can only be lower than $mscf(l)$ if $mscf(l) \neq -\infty$, this means that the second case in the definition of $mscf$ in the statement of this theorem applies for l and we have $cost'(l) < \max_{a \xrightarrow{l} b \in T_{fin}^h} (h_{\mathcal{T}_h}^*(a, cost) - h_{\mathcal{T}_h}^*(b, cost))$. Therefore, there exists an abstract transition $a \xrightarrow{l} b \in T_{fin}^h$ with $cost'(l) < h_{\mathcal{T}_h}^*(a, cost) - h_{\mathcal{T}_h}^*(b, cost)$. This implies $h_{\mathcal{T}_h}^*(a, cost) > cost'(l) + h_{\mathcal{T}_h}^*(b, cost)$. Because $h(s, cost') = h(s, cost)$ for all concrete states $s \in S(\mathcal{T})$, we also have $h_{\mathcal{T}_h}^*(c, cost') = h_{\mathcal{T}_h}^*(c, cost)$ for all abstract states $c \in S(\mathcal{T}_h)$. With this, we get $h_{\mathcal{T}_h}^*(a, cost') > cost'(l) + h_{\mathcal{T}_h}^*(b, cost')$, which violates the triangle inequality for shortest paths in graphs. \square

The minimum saturated cost function can be computed with negligible overhead during the construction of pattern databases (Culberson and Schaeffer 1998; Edelkamp 2001), Cartesian abstractions (see Section 4.1) and merge-and-shrink abstractions not using *label reduction* (Sievers et al. 2014).¹

When computing saturated cost functions for abstraction heuristics we use a minor optimization. In addition to ignoring states from which there is no path to a goal state, we also ignore states (and their outgoing transitions) that are unreachable from the initial state. Since we want to apply the resulting heuristic to forward search, heuristic values of unreachable states are irrelevant, and using lower costs is always preferable if we can still preserve the heuristic values of all reachable states.

Figure 5.1 demonstrates how to compute the minimum saturated cost function (shown in the table on the right) for an abstraction heuristic (the underlying abstract transition system is depicted on the left). For example, since operator o_4 only induces a self-looping transition in the abstract transition system, its minimum saturated cost is 0, reflecting the intuition that o_4 contributes nothing to the solution under this abstraction. In contrast,

¹ Computing the minimum saturated cost function is more expensive for merge-and-shrink heuristics using label reduction. In the final abstract transition system \mathcal{T} of a merge-and-shrink heuristic using label reduction all transitions with the same weight share the same label. Consequently, \mathcal{T} does not hold the information which original label induces which abstract transitions. To compute the minimum saturated cost function for \mathcal{T} we therefore need to compute the set of transitions that each original label induces in \mathcal{T} . This requires knowing the preimage of each abstract state in \mathcal{T} . For merge-and-shrink heuristics using linear merge strategies, we can represent the preimage of an abstract state as a binary decision diagram (BDD). Computing whether there is a transition with a given label between two abstract states represented as BDDs is expensive, but polynomial. For non-linear merge strategies we can represent each preimage as a sentential decision diagram (SDD), but it is unknown whether the corresponding test runs in polynomial time.

Algorithm 6 Interleaved saturated cost partitioning algorithm. Given a cost function $cost$, it computes a set of abstraction heuristics and corresponding minimum saturated cost functions that form a cost partitioning.

```

1: procedure INTERLEAVEDSATURATEDCOSTPARTITIONING( $cost$ )
2:   while not TERMINATIONCONDITION do
3:      $\mathcal{T}_h \leftarrow$  compute abstract transition system using  $cost$ 
4:      $h \leftarrow$  abstraction heuristic corresponding to  $\mathcal{T}_h$ 
5:      $m_{scf} \leftarrow$  minimum saturated cost function for  $h$  and  $cost$ 
6:      $cost \leftarrow cost - m_{scf}$ 

```

operator o_1 induces two transitions and we need to take both of them into account when computing the minimum saturated cost to ensure that no goal distance changes.

We use Theorem 5.1 to define a saturator that always returns the minimum saturated cost function for abstraction heuristics.

Definition 5.4 Minimum Saturator for Abstraction Heuristics.

Let h be an abstraction heuristic. The function $saturate_h : \mathcal{C}(\mathcal{T}_h) \rightarrow \mathcal{C}(\mathcal{T}_h)$ with $saturate_h(cost) = m_{scf}$, where m_{scf} is the minimum saturated cost function for h and $cost$, is the minimum saturator for h .

In all experiments, we use the minimum saturator $saturate_h$ from Definition 5.4 for abstraction heuristics h .

Example We use the two abstraction heuristics h_1 and h_2 from Figure 3.1 on p. 15 to show a complete run of the saturated cost partitioning algorithm. After choosing the order $\langle h_1, h_2 \rangle$ and the minimum saturators $\langle saturate_{h_1}, saturate_{h_2} \rangle$ (see Definition 5.4), we are ready to compute the saturated cost partitioning. The first remaining cost function is $remain_0 = cost = \langle 4, 1, 4, 1 \rangle$. Under $remain_0$ the abstract goal distances of the three abstract states in \mathcal{T}_{h_1} are 5, 1 and 0. The minimum saturated cost function $saturate_1(h_1, remain_0) = \langle 4, 0, 1, 1 \rangle$ tells us that we can decrease the cost for operators o_2 and o_3 in h_1 without affecting any goal distances. Subtracting the saturated costs from $remain_0$ yields our new remaining cost function $remain_1 = \langle 0, 1, 3, 0 \rangle$. Under $remain_1$ the goal distances in \mathcal{T}_{h_2} are 3, 3 and 0 and we have $saturate_2(h_2, remain_1) = \langle 0, 0, 3, 0 \rangle$. The two saturated cost functions form a cost partitioning and we have $h_{\langle h_1, h_2 \rangle}^{SCP}(s_1, cost) = 5 + 3 = 8$. Note that the cost of operator o_2 is not needed to justify this estimate (i.e., $remain_2 = \langle 0, 1, 0, 0 \rangle$) and we could use it for other heuristics.

5.4 Interleaved Saturated Cost Partitioning Algorithm

In our implementation, we exploit the fact that saturated cost partitioning only needs to hold one abstract transition system in memory at a time by interleaving abstraction computation and cost partitioning. Algorithm 6 shows pseudo-code for the procedure: starting with the original cost function $cost$, we iteratively create an abstract transition system \mathcal{T}_h using $cost$, obtain the corresponding abstraction heuristic h , compute the minimum saturated cost function $mscf$ for h and $cost$, subtract $mscf$ from $cost$ and build the next abstraction using the remaining costs.

The procedure can terminate after computing a given number of abstractions or once no further useful abstractions can be found. The sequence of saturated cost functions computed by the procedure then forms a cost partitioning which can be used to sum the heuristics associated with the abstractions admissibly.

MULTIPLE CARTESIAN ABSTRACTIONS

Having discussed how we can combine multiple abstraction heuristics, we now need a way to come up with different abstractions to combine. We have shown above how to build a single Cartesian abstraction using a timeout of X seconds. The simplest idea to come up with n additive abstractions, then, is to repeat the CEGAR algorithm n times with timeouts of X/n seconds, computing the saturated cost function after each iteration, and using the remaining cost in subsequent iterations.

Table 6.1 shows the number of solved tasks from our benchmark set for $X = 900$ s and different values of n . All versions are given a time limit of 1800 seconds (of which at most X seconds are used to construct the abstractions) and 3.5 GiB of memory to find a solution.

We see that building more than one abstraction is mildly detrimental. In almost all domains coverage remains the same or decreases slightly when increasing the number of abstractions. Out of 40 domains, only Airport and Logistics benefit from using 100–500 and 5–100 abstractions, respectively, but, as in the other domains, using more abstractions makes coverage decrease again.

Overall, we note that computing multiple abstractions is not beneficial. We hypothesize that this is the case because the computed abstractions are too similar to each other, focusing mostly on the same parts of the task. Computing multiple abstractions does not yield a more informed additive heuristic and instead just consumes time that could have been used to further refine a single abstraction.

To see why diversifying abstractions is essential, consider the extreme case where we evaluate the same abstraction heuristic h under two different cost functions $cost_1$ and $cost_2$. For every state s we have $h(s, cost_1) + h(s, cost_2) \leq h(s, cost_1 + cost_2)$, i.e., using the sum of heuristic values is *dominated* by using h only once with cost function $cost_1 + cost_2$. (This follows from the admissibility of cost partitioning.) So we need to make sure that the abstractions computed in different iterations of the algorithm are sufficiently different.

There are several possible ways of ensuring such diversity within the CEGAR framework. One way is to make sure that different iterations of the CEGAR algorithm produce different results even when presented with the same input planning task. This is quite possible to do because the CEGAR algorithm has several choice points that affect its outcome, in particular in the refinement step where there are frequently multiple splits to choose

Abstractions	1	5	10	50	100	500	1000	5000
airport ₍₅₀₎	22	22	22	22	23	23	22	22
driverlog ₍₂₀₎	11	10	10	10	10	10	10	10
logistics ₍₆₃₎	17	18	18	18	18	17	16	16
mprime ₍₃₅₎	27	27	27	27	26	27	26	25
mystery ₍₃₀₎	18	18	18	17	17	17	17	17
nomystery ₍₂₀₎	10	10	10	9	9	9	9	8
parcprinter ₍₅₀₎	22	21	20	18	18	18	18	18
pipes-t ₍₅₀₎	14	14	14	14	14	13	13	13
rovers ₍₄₀₎	7	6	6	6	6	6	6	6
sokoban ₍₅₀₎	43	43	43	43	43	43	43	41
trucks ₍₃₀₎	8	8	7	6	6	6	6	6
woodwork ₍₅₀₎	15	15	15	13	13	13	13	13
zenotravel ₍₂₀₎	9	9	9	8	8	8	8	8
...
Sum ₍₁₆₆₇₎	706	704	702	694	694	693	690	684

Table 6.1: Number of solved tasks for a growing number of Cartesian abstractions. We omit domains in which coverage does not change and highlight best results in bold.

from. By ensuring that these choices are resolved differently in different iterations of the algorithm, we can achieve some degree of diversification. We call this approach *diversification by refinement strategy*.

Another way of ensuring diversity, even in the case where the CEGAR algorithm always generates the same abstraction when faced with the same input task, is to modify the inputs to the CEGAR algorithm. Rather than feeding the actual planning task to the CEGAR algorithm, we can present it with different “subproblems” in every iteration, so that it will naturally generate different results. To ensure that the resulting heuristic is admissible, it is sufficient that every subproblem we use as an input to the CEGAR algorithm is itself an abstraction of the original task. We call this approach *diversification by task modification*. We discuss these two approaches in the following sections.

6.1 Diversification by Refinement Strategy

As discussed in Section 4.3, when handling a flaw $\langle s, c \rangle$, there are often multiple variables $\mathcal{V}' \subseteq \mathcal{V}$ for which a split is feasible. Our first diversification method changes how CEGAR chooses one of these variables. It bases this decision on the h^{add} values (Bonet and Geffner 2001) of the values in $\text{dom}(v, c)$. More concretely, it chooses among the feasible candidates by selecting the variable that has a value in c with the highest h^{add} value, i.e., the variable $v \in \mathcal{V}'$ that maximizes $\max_{v \mapsto x \in \text{dom}(v, c)} h^{\text{add}}(v \mapsto x)$.

Abstractions	1	5	10	50	100	500	1000	5000
airport ⁽⁵⁰⁾	22	23	23	24	25	25	25	25
blocks ⁽³⁵⁾	20	20	19	18	18	18	18	18
driverlog ⁽²⁰⁾	10	10	10	10	10	10	10	9
floortile ⁽⁴⁰⁾	3	2	2	2	2	2	2	2
ged ⁽²⁰⁾	17	16	15	15	15	15	15	15
logistics ⁽⁶³⁾	20	24	22	23	22	22	23	22
miconic ⁽¹⁵⁰⁾	55	61	60	60	60	59	59	58
mprime ⁽³⁵⁾	28	27	26	25	25	26	25	24
mystery ⁽³⁰⁾	17	18	17	17	17	17	18	17
nomystery ⁽²⁰⁾	10	14	13	10	10	11	10	13
parcprinter ⁽⁵⁰⁾	20	20	20	20	20	20	22	22
pipes-nt ⁽⁵⁰⁾	19	18	18	18	18	18	19	19
rovers ⁽⁴⁰⁾	6	8	8	8	8	8	8	7
tidybot ⁽⁴⁰⁾	23	26	29	26	25	27	25	23
tpp ⁽³⁰⁾	6	7	7	7	7	6	7	7
trucks ⁽³⁰⁾	8	9	9	9	9	9	9	9
visitall ⁽⁴⁰⁾	14	14	13	13	13	13	13	13
woodwork ⁽⁵⁰⁾	15	15	15	17	17	14	15	16
...
Sum ⁽¹⁶⁶⁷⁾	707	726	720	716	715	714	717	713

Table 6.2: Number of solved tasks for a growing number of Cartesian abstractions preferring to refine atoms with high h^{add} values. We omit domains in which coverage does not change and highlight best results in bold.

This “max- h^{add} ” refinement strategy (unlike the default strategy “max-refined”) is affected by the costs of the operators, which change from iteration to iteration as costs are used up by previously computed abstractions. This inherently biases CEGAR towards regions of the state space where operators still have high costs.

Table 6.2 shows the results for this approach. We see that the h^{add} -based refinement strategy leads to better results than the default refinement strategy on average: while both methods solve about the same number of tasks in the basic case of only one abstraction (706 tasks for the default strategy vs. 707 tasks for h^{add} -based refinement), for values of n between 5 and 5000 we obtain 18–29 additional solved tasks compared to the corresponding columns in Table 6.1. We also see that all tested values of n lead to a higher total coverage compared to a single abstraction, whereas the total coverage score of the original refinement strategy never benefits from using more than one abstraction. The h^{add} -based refinement strategy obtains the maximum number of solved tasks with 5 abstractions (726 tasks). Using more than that leads to a decrease in total coverage.

Overall, we see that using a refinement strategy that takes into account the operator costs and hence interacts well with cost partitioning can lead

to better scalability for additive CEGAR heuristics. However, the improvements obtained in this way are quite modest. This motivates diversification by task modification, which is a somewhat more drastic approach than diversification by refinement strategy. The basic idea is that we identify different aspects of the planning task and then generate an abstraction of the original task for each of these aspects. Each invocation of the CEGAR algorithm uses one of these abstractions as its input and is thus constrained to exclusively focus on one aspect.

We propose two different ways for coming up with such “focused subproblems”: *abstraction by goals* and *abstraction by landmarks*.

6.2 Abstraction by Goals

Our first approach, abstraction by goals, generates one abstract task for each goal atom of the planning task. The number of abstractions generated is hence equal to the number of goals.

If $v \mapsto d$ is a goal atom, we create a modified planning task which is identical to the original one except that $v \mapsto d$ is the only goal atom. This means that the original and modified task have exactly the same states and transitions and only differ in their goal states: in the original task, *all* goals need to be satisfied in a goal state, but in the modified one, *only* $v \mapsto d$ needs to be reached. The goal states of the modified task are hence a superset of the original goal states, and we can conclude that the modification defines a (non-induced) abstraction in the sense of Definition 2.9 (where the abstraction mapping is the identity function).

Abstracting by goals has the obvious drawback that it only works for tasks with more than one goal atom. Since any task could potentially be reformulated to only contain a single goal atom, a smarter way of diversification is desirable.

6.3 Abstraction by Landmarks

Our next diversification strategy solves this problem by using *fact landmarks* instead of goal atoms to define subproblems of a task. Fact landmarks are atoms that have to be true at least once in all plans for a given task (e.g., Hoffmann et al. 2004). Since obviously all goal atoms are also fact landmarks, this method can be seen as a generalization of the previous strategy.

More specifically, we generate the causal fact landmarks of the delete relaxation of the planning task Π with the algorithm by Keyder et al. (2010) for finding h^m landmarks with $m = 1$. Then for each landmark $l = v \mapsto d$ we compute a modified task Π_l that considers l as the only goal atom.

Algorithm 7 Construct modified task for landmark $v \mapsto d$.

```

1: function LANDMARKTASK( $\Pi, v \mapsto d$ )
2:    $\langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle \leftarrow \Pi$ 
3:    $\mathcal{V}' \leftarrow \mathcal{V}$ 
4:    $F \leftarrow \text{POSSIBLYBEFORE}(\Pi, v \mapsto d)$ 
5:   for all  $v' \in \mathcal{V}'$  do
6:      $\text{dom}(v') \leftarrow \{d' \in \text{dom}(v') \mid v' \mapsto d' \in F \cup \{v \mapsto d\}\}$ 
7:    $\mathcal{O}' \leftarrow \{o \in \mathcal{O} \mid \text{pre}(o) \subseteq F\}$ 
8:   for all  $o \in \mathcal{O}'$  do
9:     if  $v \mapsto d \in \text{eff}(o)$  then
10:       $\text{eff}(o) \leftarrow \{v \mapsto d\}$ 
11:   return  $\langle \mathcal{V}', \mathcal{O}', s_0, \{v \mapsto d\} \rangle$ 

12: function POSSIBLYBEFORE( $\Pi, v \mapsto d$ )
13:    $\langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle \leftarrow \Pi$ 
14:    $F \leftarrow s_0$ 
15:   while  $F$  has not reached a fixpoint do
16:     for all  $o \in \mathcal{O}$  do
17:       if  $v \mapsto d \notin \text{eff}(o) \wedge \text{pre}(o) \subseteq F$  then
18:          $F \leftarrow F \cup \text{eff}(o)$ 
19:   return  $F$ 

```

Without further modifications, however, this change does not constitute an abstraction (not even a non-induced abstraction), and hence the resulting heuristic could be inadmissible. This is because landmarks do not have the same semantics as goals: goals need to be satisfied *at the end* of a plan, but landmarks are only required *at some point* during the execution of a plan.

Existing landmark-based heuristics address this difficulty by remembering which landmarks have been achieved en route to any given state and only base the heuristic information on landmarks which have not yet been achieved (e.g., Karpas and Domshlak 2009; Richter et al. 2008). This makes these heuristics *path-dependent*: their heuristic values are no longer a function of the state alone.

Path-dependency comes at a significant memory cost for storing landmark information, so we propose an alternative approach that is purely state-based. For every state s , we use a sufficient criterion for deciding whether the given landmark *might have been achieved* on the path from the initial state to s . If yes, s is considered as a goal state in the modified task and hence will be assigned a heuristic value of 0 by the associated abstraction heuristic.

Without path information, how can we decide whether a given landmark could have been reached prior to state s ? The key to this question is the notion of a *possibly-before* set for atoms of delete relaxations, which has been previously considered by Porteous and Cresswell (2002). We say that an atom f' is *possibly before* atom f if f' can be achieved in the delete relaxation of the planning task without achieving f . We write $pb(f)$ for the set of atoms that are possibly before f ; this set can be computed using a fixpoint computation shown in Algorithm 7 (function POSSIBLYBEFORE). This simple implementation runs in time $O(n^2)$ for a task of size n , but a linear time version is possible by using suitable data structures. From the monotonicity properties of delete relaxations, it follows that if l is a delete-relaxation landmark and all atoms of the current state s are contained in $pb(l)$, then l still has to be achieved from s .

Based on this insight, we can construct Π_l as follows. First, we compute $pb(l)$. The modified task Π_l only contains the atoms in $pb(l)$ and l itself; all other atoms are removed. The landmark l is the only goal. The initial state and operators are identical to the original task, except that we discard operators whose preconditions are not contained in $pb(l)$ (by the definition of possibly-before sets, these can only become applicable after reaching l) and for all operators that achieve l , we make l their only effect. (Adapting such operators is necessary because they might have other effects that fall outside $pb(l)$. Note that such operators are guaranteed to achieve a goal state, and for an abstraction heuristic it does not matter which exact goal state we end up in.) The complete construction is shown in Algorithm 7.

The states $S(\Pi_l)$ of the modified task are exactly the states s of the original planning task where $s \subseteq pb(l) \cup \{l\}$. The abstraction function that is associated with the modified task maps every state in $S(\Pi_l)$ to itself. In all other states the landmark might potentially have been achieved, so they should be mapped to an arbitrary goal state of the modified task. This mapping is easy to represent within the framework of Cartesian abstraction because $S(\Pi_l)$ is a Cartesian set and its complement can be represented as the disjoint union of a small number of Cartesian sets (bounded by the number of variables of the planning task). Like in the case of abstraction by goals, the goal states of the landmark task are a superset of the goal states in the original task. Therefore, the resulting abstraction is not induced in the sense of Definition 2.9.

In detail, we establish the mapping by refining the abstract transition system before we enter the refinement loop. Starting from the trivial abstraction with a single abstract state, we iterate over all variables v . In the iteration for variable v , we split the (then current) abstract initial state into states a and b with $dom(v, a) = dom(v) \cap (pb(l) \cup \{l\})$ and $dom(v, b) = dom(v) \setminus (pb(l) \cup \{l\})$. Afterwards, we declare all states in the abstract transition system (including the initial state) as goal states.

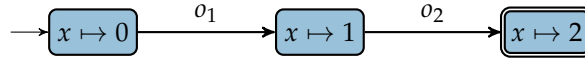


Figure 6.1: Example task in which operators o_1 and o_2 change the value of the single variable x from its initial value 0 to 1 and from 1 to its desired value 2.

6.4 Abstraction by Landmarks: Improved

In the basic form just presented, the tasks constructed for fact landmarks do not provide as much diversification as we would desire. We illustrate the issue with the example task depicted in Figure 6.1. The task has three landmarks $x \mapsto 0$, $x \mapsto 1$ and $x \mapsto 2$ that must be achieved in exactly this order in every plan. When we compute the abstraction for $x \mapsto 1$, the CEGAR algorithm has to find a plan for getting from $x \mapsto 0$ to $x \mapsto 1$. Similarly, the abstraction procedure for $x \mapsto 2$ has to return a solution that takes us from $x \mapsto 0$ to $x \mapsto 2$. Since going from $x \mapsto 0$ to $x \mapsto 2$ includes the subproblem of going from $x \mapsto 0$ to $x \mapsto 1$, we have to find a plan from $x \mapsto 0$ to $x \mapsto 1$ twice, which runs counter to our objective of finding abstractions that focus on different aspects on the planning task.

To alleviate this issue, we propose an alternative construction for the planning task for landmark l . The key idea is that we employ a further abstraction that reflects the intuition that at the time we achieve l , certain other landmarks have already been achieved.

In detail, the alternative construction proceeds as follows. We start by performing the basic landmark task construction described in Algorithm 7, resulting in a planning task for landmark l which we denote by Π_l .

Furthermore, we use a sound algorithm (Keyder et al. 2010) for computing *natural* and *greedy necessary landmark orderings* (e.g., Hoffmann et al. 2004; Richter et al. 2008) to determine a set L' of landmarks that must necessarily be achieved before l . Note that, unless l is a landmark that is already satisfied in the initial state (a trivial case we can ignore because the Cartesian abstraction heuristic is identical to 0 in this case), L' contains at least one landmark for each variable of the planning task because initial state atoms are landmarks that must be achieved before l .

Finally, we perform a *domain abstraction* (Hernádvölgyi and Holte 2000) that combines, for each variable v' , all the atoms $v' \mapsto d' \in L'$ based on the same variable into a single atom.

For example, consider the landmark $l = x \mapsto 2$ in the above example. We detect that $x \mapsto 0$ and $x \mapsto 1$ are landmarks that must be achieved before l . They both refer to the variable x , so we combine the values 0 and 1 into a single value. The effect of this is that in the task for l , we no longer

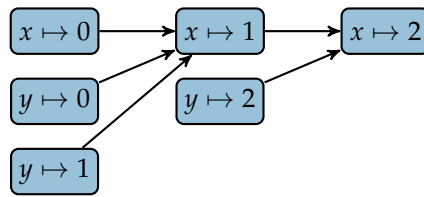


Figure 6.2: Example landmark ordering. To avoid focusing on the same parts of the task in multiple subtasks the improved abstraction-by-landmarks procedure combines the atoms $y \mapsto 0$ and $y \mapsto 1$ before building the abstraction for $x \mapsto 1$. For the subtask $x \mapsto 2$, it combines the atoms $y \mapsto 0, y \mapsto 1, y \mapsto 2$ in one group and $x \mapsto 0$ and $x \mapsto 1$ in another.

need to find a subplan from $x \mapsto 0$ to $x \mapsto 1$. Figure 6.2 illustrates the procedure with a slightly more complex example.

EXPERIMENTAL EVALUATION

We evaluate five different ways of creating Cartesian abstraction heuristics with counterexample-guided abstraction refinement: a single abstraction (h^{CEGAR} , Section 4.2), abstraction by goals ($h_{s_*}^{\text{CEGAR}}$, Section 6.2), abstraction by landmarks ($h_{\text{LM}}^{\text{CEGAR}}$, Section 6.3), improved abstraction by landmarks ($h_{\text{LM}^+}^{\text{CEGAR}}$, Section 6.4), and a combination of the latter two methods ($h_{\text{LM}+s_*}^{\text{CEGAR}}$), which we describe below.

In Part II we analyze how the order in which saturated cost partitioning considers the component heuristics influences the resulting cost-partitioned heuristic. For now, we let the saturated cost partitioning algorithm consider the subtasks in a randomized order.

In addition to comparing the resulting heuristics to each other, we contrast them to some of the strongest abstraction heuristics from the literature:

- h^{iPDB} : the canonical heuristic using pattern databases found by 15 minutes of hill climbing (Haslum et al. 2007; Sievers et al. 2012)
- $h^{\text{M\&S}}$: merge-and-shrink using bisimulation, the SCC-DFP merge strategy and at most 50 000 abstract states (Helmert et al. 2014; Sievers et al. 2016)
- $h_{\text{SYS}}^{\text{PhO}}$: post-hoc optimization using systematic patterns of sizes 1 and 2 (Pommerening et al. 2013)

We let all versions that use CEGAR refine for at most 15 minutes. For the additive CEGAR versions we distribute the refinement time equally among the abstractions. Table 7.1 shows the number of solved instances for the compared heuristics.

7.1 Comparison of CEGAR to Other Abstraction Heuristics

We begin our analysis by comparing the basic h^{CEGAR} heuristic based on a single abstraction to the heuristics from the literature. While the total coverage of h^{CEGAR} (706 tasks) is lower than that of h^{iPDB} (881 tasks), $h^{\text{M\&S}}$ (808 tasks) and $h_{\text{SYS}}^{\text{PhO}}$ (737 tasks), it outperforms all of them in Mystery and MPrime. Table 7.2 compares the heuristics on a per-domain basis and shows that h^{CEGAR} solves more tasks than h^{iPDB} , $h^{\text{M\&S}}$ and $h_{\text{SYS}}^{\text{PhO}}$ in 2, 7 and 14 domains, respectively. However, the opposite is true for 27, 23 and 13 domains.

	h^{iPDB}	$h^{\text{M\&S}}$	h^{PhO}	h^{CEGAR}	$h_{s_*}^{\text{CEGAR}}$	$h_{\text{LM}}^{\text{CEGAR}}$	$h_{\text{LM}^+}^{\text{CEGAR}}$	$h_{\text{LM}^+s_*}^{\text{CEGAR}}$
airport ⁽⁵⁰⁾	38	18	27	22	33	32	28	33
barman ⁽³⁴⁾	4	4	4	4	4	4	4	4
blocks ⁽³⁵⁾	28	28	26	18	18	18	18	18
childsnaack ⁽²⁰⁾	0	0	0	0	0	0	0	0
depot ⁽²²⁾	11	7	7	5	5	7	7	7
driverlog ⁽²⁰⁾	13	13	13	11	13	13	13	13
elevators ⁽⁵⁰⁾	41	31	36	33	39	39	37	39
floortile ⁽⁴⁰⁾	2	6	2	2	2	2	2	2
freecell ⁽⁸⁰⁾	21	20	15	20	20	20	37	36
ged ⁽²⁰⁾	19	19	15	15	15	15	15	15
grid ⁽⁵⁾	3	2	2	2	2	2	2	3
gripper ⁽²⁰⁾	8	20	7	8	8	8	8	8
hiking ⁽²⁰⁾	12	13	11	12	12	12	13	13
logistics ⁽⁶³⁾	31	25	26	17	26	26	22	26
miconic ⁽¹⁵⁰⁾	69	79	54	55	63	66	71	72
movie ⁽³⁰⁾	30	30	30	30	30	30	30	30
mprime ⁽³⁵⁾	24	23	21	27	27	27	27	27
mystery ⁽³⁰⁾	17	17	15	18	18	18	18	18
nomystery ⁽²⁰⁾	20	18	16	10	18	14	13	14
openstacks ⁽¹⁰⁰⁾	49	49	47	49	49	49	49	49
parcprinter ⁽⁵⁰⁾	38	45	30	22	24	22	24	24
parking ⁽⁴⁰⁾	13	6	3	0	0	0	0	0
pathways ⁽³⁰⁾	4	4	4	4	4	4	4	4
pegsol ⁽⁵⁰⁾	48	48	44	44	44	44	44	44
pipes-nt ⁽⁵⁰⁾	21	18	15	17	17	17	18	18
pipes-t ⁽⁵⁰⁾	18	16	9	14	14	13	14	14
psr-small ⁽⁵⁰⁾	50	50	49	49	49	49	49	49
rovers ⁽⁴⁰⁾	8	8	7	7	7	7	7	7
satellite ⁽³⁶⁾	6	7	6	6	7	7	7	7
scanalyzer ⁽⁵⁰⁾	23	23	11	21	21	21	21	21
sokoban ⁽⁵⁰⁾	50	47	49	43	41	41	41	41
storage ⁽³⁰⁾	16	15	15	15	16	16	16	16
tetris ⁽¹⁷⁾	10	2	3	9	9	9	9	9
tidybot ⁽⁴⁰⁾	22	1	21	22	28	26	28	28
tpp ⁽³⁰⁾	6	8	6	6	10	7	7	7
transport ⁽⁷⁰⁾	35	24	21	24	24	24	24	24
trucks ⁽³⁰⁾	9	7	7	8	11	10	12	12
visitall ⁽⁴⁰⁾	28	13	27	13	13	13	13	13
woodwork ⁽⁵⁰⁾	23	32	25	15	21	21	21	21
zenotravel ⁽²⁰⁾	13	12	11	9	12	12	12	12
Sum ⁽¹⁶⁶⁷⁾	881	808	737	706	774	765	785	798

Table 7.1: Number of solved tasks by domain for different heuristics. We highlight best values in bold.

	h^{iPDB}	$h^{M\&S}$	h^{PhO}	h^{CEGAR}	$h^{CEGAR}_{s^*}$	h^{CEGAR}_{LM}	h^{CEGAR}_{LM+}	$h^{CEGAR}_{LM+s^*}$	Coverage
h^{iPDB}	-	19	31	27	24	24	22	21	881
$h^{M\&S}$	8	-	24	23	17	18	17	16	808
h^{PhO}	1	7	-	13	7	7	8	7	737
h^{CEGAR}	2	7	14	-	1	2	1	1	706
$h^{CEGAR}_{s^*}$	6	10	21	14	-	7	5	2	774
h^{CEGAR}_{LM}	6	9	20	14	2	-	4	0	765
h^{CEGAR}_{LM+}	9	9	20	18	6	8	-	1	785
$h^{CEGAR}_{LM+s^*}$	9	11	21	19	7	10	6	-	798

Table 7.2: Left: Pairwise comparison of different abstraction heuristics. The entry in row r and column c holds the number of domains in which heuristic r solved more tasks than heuristic c . For each heuristic pair we highlight the maximum of the entries $\langle r, c \rangle$ and $\langle c, r \rangle$. Right: Total number of solved tasks by each heuristic.

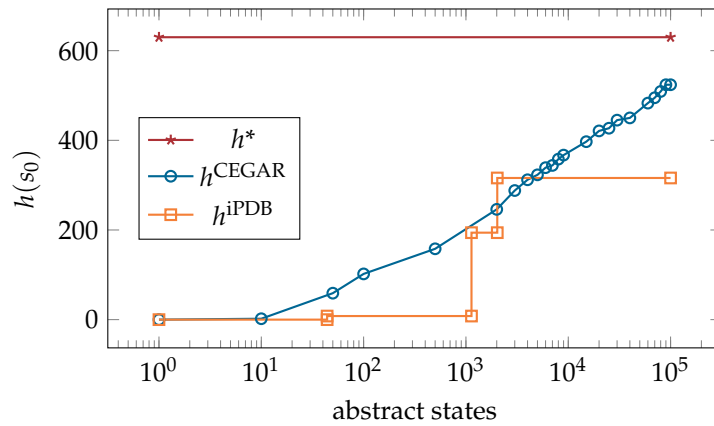


Figure 7.1: Initial state heuristic values for transport-o8 #23.

Although h^{CEGAR} is outperformed by h^{iPDB} , the former still has a theoretical advantage over the latter. While the CEGAR loop is guaranteed to converge to a solution, h^{iPDB} can get stuck in local minima. Also, the h^{CEGAR} estimates tend to grow much more smoothly towards the perfect estimates than the ones by h^{iPDB} . Figure 7.1 illustrates this by comparing how the cost estimates for the initial state grow with the number of abstract states on an example task. The h^{CEGAR} estimates are generally higher than those of h^{iPDB} . This behavior can be observed in many domains.

7.2 Comparison of CEGAR Heuristics

Comparing the results for h^{CEGAR} and $h_{s_*}^{\text{CEGAR}}$, we see that decomposing the task by goals and finding multiple abstractions separately instead of using only a single abstraction raises the number of solved tasks from 706 to 774. This substantial improvement is due to the fact that 14 domains profit from using $h_{s_*}^{\text{CEGAR}}$ while coverage decreases in only one domain (see Table 7.2).

Decomposing by goals has a slight edge over landmarks decompositions: $h_{s_*}^{\text{CEGAR}}$ solves 774 tasks while $h_{\text{LM}}^{\text{CEGAR}}$ has a coverage score of 765. Also for the individual domains $h_{s_*}^{\text{CEGAR}}$ is usually preferable: $h_{s_*}^{\text{CEGAR}}$ solves more tasks than $h_{\text{LM}}^{\text{CEGAR}}$ in 7 domains, while the opposite is true in 2 domains. However, when we employ the improved $h_{\text{LM}^+}^{\text{CEGAR}}$ heuristic, which uses domain abstraction to avoid duplicate work during the refinement process, the number of solved problems increases to 785.

Abstraction by Landmarks and Goals In Table 7.2, we can observe that $h_{s_*}^{\text{CEGAR}}$ and $h_{\text{LM}^+}^{\text{CEGAR}}$ outperform each other in multiple domains: for maximum coverage $h_{s_*}^{\text{CEGAR}}$ is preferable in 5 domains, whereas $h_{\text{LM}^+}^{\text{CEGAR}}$ should be preferred in 6 domains. This suggests trying to combine the two approaches.

We do so by first computing abstractions for all subproblems returned by the *abstraction by landmarks* method. If afterwards the refinement time has not been consumed, we also calculate abstractions for the subproblems returned by the *abstraction by goals* decomposition strategy for the remaining time. Not only does this approach ($h_{\text{LM}^+s_*}^{\text{CEGAR}}$) solve as many problems as the better performing ingredient technique in many individual domains, but it often even outperforms both original diversification methods, raising the total number of solved tasks to 798.

Comparing $h_{\text{LM}^+s_*}^{\text{CEGAR}}$ to h^{CEGAR} , we see that $h_{\text{LM}^+s_*}^{\text{CEGAR}}$ solves 92 more tasks than h^{CEGAR} (798 vs. 706). This difference in coverage is substantial because in most domains solving an additional task optimally becomes exponentially more difficult as the tasks get larger.

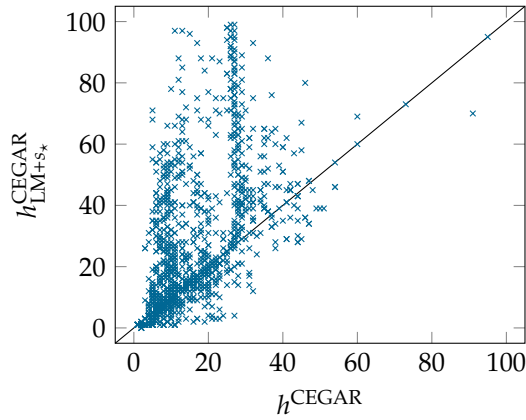


Figure 7.2: Comparison of the heuristic estimates for the initial state made by h^{CEGAR} and $h^{\text{CEGAR}}_{\text{LM}+s_*}$. For better visibility, we only include tasks for which both heuristics estimate the solution cost to be at most 100. Points above the diagonal represent tasks for which $h^{\text{CEGAR}}_{\text{LM}+s_*}$ yields a better estimate than h^{CEGAR} .

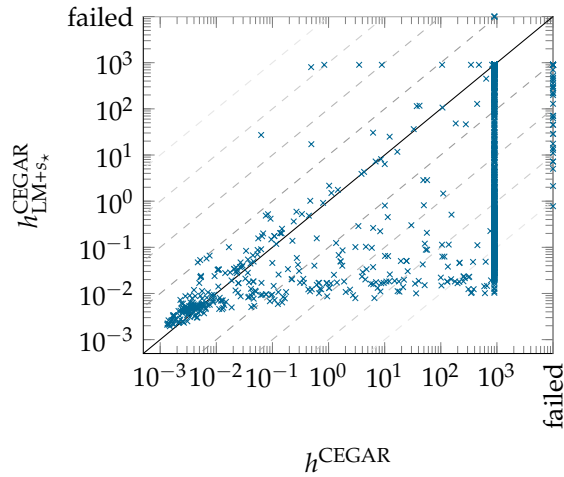


Figure 7.3: Comparison of the time (in seconds) taken by h^{CEGAR} and $h^{\text{CEGAR}}_{\text{LM}+s_*}$ to compute all abstractions on the benchmark tasks. Failed runs exceeded the memory limit of 3.5 GiB. Points below the diagonal represent tasks for which $h^{\text{CEGAR}}_{\text{LM}+s_*}$ needs less time for the computation than h^{CEGAR} .

The big increase in coverage can be explained by the fact that for the majority of tasks $h_{LM+s_*}^{CEGAR}$ estimates the solution cost much better than h^{CEGAR} , as shown in Figure 7.2. One might expect that the increased informativeness would come with a time penalty, but in Figure 7.3 we can see that in fact $h_{LM+s_*}^{CEGAR}$ takes *less* time to compute the abstractions than h^{CEGAR} . If the CEGAR refinement loop does not exceed the memory limit, it only stops if it runs out of time or finds a concrete solution. Figure 7.3 therefore tells us that for most tasks h^{CEGAR} does not find a solution early, but instead uses the full 15 minutes for the refinement whereas $h_{LM+s_*}^{CEGAR}$ almost always needs less time. Due to building many small abstractions instead of a single large one, $h_{LM+s_*}^{CEGAR}$ also uses less memory than h^{CEGAR} and therefore exceeds the memory limit in fewer cases.

While h^{CEGAR} has a lower total coverage than the three abstraction heuristics from the literature, $h_{LM+s_*}^{CEGAR}$ outperforms h_{Sys}^{PhO} regarding total coverage. In Table 7.2 we see that $h_{LM+s_*}^{CEGAR}$ solves more tasks than h^{iPDB} , $h^{M\&S}$ and h_{Sys}^{PhO} in 9, 11 and 21 domains, respectively. The opposite is true in 21, 16 and 7 domains. In the next part, we show that non-random heuristic orders make saturated cost partitioning heuristics much stronger. Additionally, we demonstrate that maximizing over heuristics for multiple orders further increases heuristic accuracy significantly. These changes let saturated cost partitioning heuristics over Cartesian abstractions of landmark and goal task decompositions outperform all three abstraction heuristics from the literature (see column h_{div}^{SCP} -CART in Table A.6 in the appendix).

Part II

ORDERS FOR SATURATED COST PARTITIONING

Since saturated cost partitioning distributes costs greedily, it is highly susceptible to the order in which it considers the component heuristics. We propose a greedy algorithm for finding orders and also show how to optimize orders with a hill-climbing search. Both techniques lead to significantly better heuristic estimates than using random orders. Moreover, using multiple orders results in a heuristic that is significantly better informed than any single-order heuristic, especially when we actively search for diverse orders.

SINGLE ORDERS

In Chapter 5, we introduced the saturated cost partitioning algorithm. It assigns costs greedily and is therefore affected by the order in which the heuristics are considered. In this chapter, we show that the choice of order has a strong impact on saturated cost partitioning, both theoretically and experimentally.

Component Heuristics For the experiments in Chapters 6 and 7, we computed the abstractions “just-in-time”, guiding the computation of abstractions by the current remaining cost function. In this chapter, we want to evaluate the impact of different heuristic orderings on saturated cost partitioning. Therefore, we need to ensure that all ordering algorithms work on the same heuristics. Consequently, in contrast to the approach above, we fix the set of heuristics before computing saturated cost partitionings to be the combination of the following families of abstraction heuristics:

- pattern databases found by hill climbing (HC):
We use the algorithm by Haslum et al. (2007) for searching via hill climbing in the space of pattern collections. The algorithm evaluates candidate patterns with the canonical heuristic.¹
- pattern databases for systematically generated patterns (Sys):
We use a procedure that generates all *interesting* patterns up to a given size (Pommerening et al. 2013). Since generating the PDBs for all interesting patterns of size 3 takes too long for many tasks, we generate all interesting patterns of sizes 1 and 2.
- Cartesian abstraction heuristics (CART):
We consider Cartesian abstractions of the landmark and goal task decompositions from Chapter 6.²

¹ To ensure that the resulting pattern collection is the same in independent algorithm runs we do not limit the time for hill climbing but instead limit the number of generated patterns by 200.

² The Cartesian abstractions underlying the heuristics in CART are very similar to the abstractions used by h_{LM+s}^{CEGAR} in Chapter 7 but not identical, due to the use of different termination criteria for the refinement loop. h_{LM+s}^{CEGAR} divides at most 15 minutes of refinement among all abstractions. To ensure that the heuristics in CART are exactly the same for every algorithm run, we do not use a time limit but instead limit the sum of non-looping transitions in all abstractions underlying the heuristics in CART by one million.

We call this combination of abstraction heuristics `COMB`, i.e., $\text{COMB} = \text{HC} \cup \text{SYS} \cup \text{CART}$. We use the combination of the heterogeneous heuristics instead of a homogeneous subset, because having more (and different) heuristics makes ordering them more difficult, which in turn makes evaluating ordering algorithms easier.

Our first analysis in this chapter shows that we can reduce the memory usage and increase the evaluation speed of saturated cost partitioning heuristics by ignoring component heuristics that contribute no information to the overall heuristic estimate.

8.1 Sparse Orders

Due to the greedy way in which saturated cost partitioning distributes costs to the heuristics, especially heuristics h that appear late in an order ω often receive a very low cost function $cost$. If all heuristic estimates of h under $cost$ are 0, we call h *useless* under $cost$.

Definition 8.1 Useless heuristics.

Let \mathcal{T} be a parameterized-cost transition system and let h be an admissible heuristic for \mathcal{T} . Then h is useless under cost function $cost \in \mathcal{C}(\mathcal{T})$ iff $h(s, cost) = 0$ for all states $s \in S(\mathcal{T})$.

Unfortunately, we can only decide whether a heuristic is useless under a given cost function after evaluating it for all states. However, for abstraction heuristics there is a sufficient condition which we can check efficiently before starting the search.

Definition 8.2 Useless abstraction heuristics.

Let h be an abstraction heuristic with associated abstract transition system \mathcal{T}_h . If $h_{\mathcal{T}_h}^*(a, cost) = 0$ for all states $a \in S(\mathcal{T}_h)$, h is useless under cost function $cost$.

If saturated cost partitioning using an order ω assigns abstraction heuristic $h \in \omega$ the cost function $cost$ and h is useless under $cost$, we can remove h from ω without changing any heuristic estimates of h_{ω}^{SCP} . Removing a heuristic means we do not have to evaluate it during search, which speeds up evaluating h_{ω}^{SCP} . For abstraction heuristics this also has the benefit of not having to store the lookup table that holds the abstract goal distances in \mathcal{T}_h under $cost$. We call orders that ignore useless heuristics *sparse orders* and all other orders *full orders*.

To evaluate the impact of using sparse orders, we compute saturated cost partitioning heuristics for random sparse orders and random full orders: $h_{\text{full}}^{\text{SCP}}$ stores all lookup tables while $h_{\text{sparse}}^{\text{SCP}}$ ignores useless pairs of heuristics and cost functions. Figure 8.1a shows the relative number of lookup tables that the two algorithms store. For the majority of tasks more than half of the lookup tables are useless. There is even a task from the

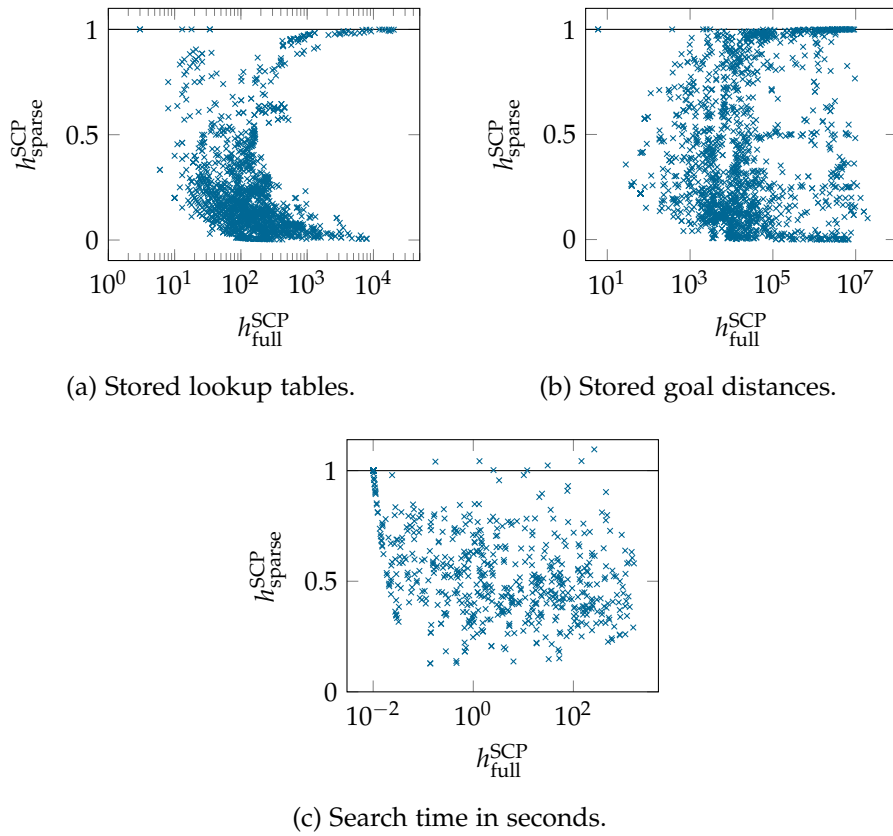


Figure 8.1: Comparing two versions of $h_{\text{random}}^{\text{SCP}}$: $h_{\text{full}}^{\text{SCP}}$ uses full orders and $h_{\text{sparse}}^{\text{SCP}}$ uses sparse orders. Each $\langle x, y \rangle$ point corresponds to a task for which $h_{\text{div}}^{\text{SCP}}$ yields a value of x and $h_{\text{sparse}}^{\text{SCP}}$ yields $x \cdot y$. Note that all x axes use a log scale.

Tetris domain where $h_{\text{full}}^{\text{SCP}}$ stores 7964 lookup tables, while $h_{\text{sparse}}^{\text{SCP}}$ determines that only 51 of them contain any information.

Since $h_{\text{sparse}}^{\text{SCP}}$ keeps much fewer lookup tables than $h_{\text{full}}^{\text{SCP}}$, it is able to represent the resulting heuristic much more compactly. Figure 8.1b shows that using sparse orders often drastically reduces the number of stored goal distances. For one task from the Floortile domain $h_{\text{full}}^{\text{SCP}}$ stores 5594881 distances whereas $h_{\text{sparse}}^{\text{SCP}}$ only keeps 363 distances. Since we use 4 bytes to store each goal distance, this means storing only 1.42 KiB instead of 21.34 MiB (ignoring the overhead of the data structures that hold the distances). Given our memory limit of 3.5 GiB, using even 100 MiB or more for storing the heuristic seems reasonable. However, the reduced memory requirements of sparse orders will be more important when we compute saturated cost partitionings for multiple orders in Chapter 9.

Not only does $h_{\text{sparse}}^{\text{SCP}}$ need less memory than $h_{\text{full}}^{\text{SCP}}$, it is also much faster to evaluate. Figure 8.1c compares the search times (without the time for computing abstractions and the saturated cost partitioning) used by the two heuristics. We can see that using sparse orders substantially reduces the search time for many tasks. As a consequence, $h_{\text{sparse}}^{\text{SCP}}$ solves two tasks (from the Openstacks and Tetris domains) for which $h_{\text{full}}^{\text{SCP}}$ runs out of time. On the 1667 tasks from our benchmark set $h_{\text{full}}^{\text{SCP}}$ finds a solution for 852 tasks and runs out of time for 443 tasks. In contrast, $h_{\text{sparse}}^{\text{SCP}}$ solves 854 tasks and fails to find a solution due to the time limit in only 99 cases, giving further evidence of its increased evaluation speed.

Due to these results, we only use sparse orders below.

8.2 Importance of Good Orders

The order in which saturated cost partitioning considers the heuristics is very important for the accuracy of the resulting cost-partitioned heuristic: two orders of the same heuristics can make the difference between a heuristic that always returns cost estimate 0 and a perfect heuristic (returning the true goal distance).

Theorem 8.1 Importance of good orders.

There exist regular fixed-cost transition systems $\langle \mathcal{T}, \text{cost} \rangle$, sets of cost-monotonic admissible heuristics \mathcal{H} for \mathcal{T} and states $s \in S(\mathcal{T})$ such that $h_{\omega}^{\text{SCP}}(s, \text{cost}) = h_{\mathcal{T}, \text{cost}}^(s, \text{cost}) > 0$ and $h_{\omega'}^{\text{SCP}}(s, \text{cost}) = 0$ for two orders $\omega, \omega' \in \Omega(\mathcal{H})$.*

Proof. Consider the example in Figure 8.2. For the concrete fixed-cost transition system $\langle \mathcal{T}, \text{cost} \rangle$ and the two abstraction heuristics h_1 and h_2 for \mathcal{T} , we have $h_{\langle h_1, h_2 \rangle}^{\text{SCP}}(s_1, \text{cost}) = 1 = h_{\mathcal{T}, \text{cost}}^*(s_1)$ and $h_{\langle h_2, h_1 \rangle}^{\text{SCP}}(s_1, \text{cost}) = 0$. \square

Note that we can arbitrarily enlarge the accuracy gap between two heuristics resulting from two different orders. In the example, it suffices to

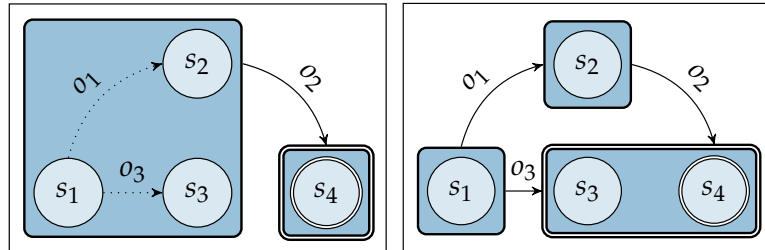


Figure 8.2: Abstraction heuristics h_1 (left) and h_2 (right) used in the proof of Theorem 8.1. The cost function is $cost = \langle 0, 1, 0 \rangle$, i.e., label o_2 has cost 1 while o_1 and o_3 have cost 0.

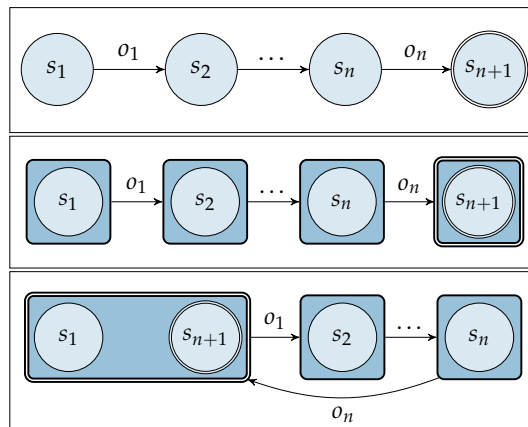


Figure 8.3: Concrete transition system \mathcal{T} (top) with abstraction heuristics h_1 (middle) and h_2 (bottom). All labels have cost 1.

adjust the cost of label o_2 . Similar instances can also be constructed for unit-cost tasks and Figure 8.3 shows an example: for the concrete fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ with $cost(l) = 1$ for all labels $l \in \mathcal{L}(\mathcal{T})$ and the two abstraction heuristics h_1 and h_2 for \mathcal{T} , we have $h_{\langle h_1, h_2 \rangle}^{SCP}(s_1) = n = h_{\mathcal{T}, cost}^*(s_1)$ and $h_{\langle h_2, h_1 \rangle}^{SCP}(s_1) = 0$. Note that this example also works if the abstraction underlying h_1 does not map each concrete state to a different abstract state.

The two examples demonstrate the importance of choosing good orders for saturated cost partitioning. There are two challenges when trying to find good orders automatically: first, we need to deal with a combinatorial search space of $n!$ possible orders for a set of n heuristics. Second, we are looking for orders that provide good guidance in all states visited during search and not only in a single state. We will deal with the second challenge later and focus on finding a good order for a single state for now.

Formally, given a fixed-cost transition system $\langle \mathcal{T}, cost \rangle$, a set of n heuristics \mathcal{H} for \mathcal{T} and a state $s \in S(\mathcal{T})$, our goal is to find an order $\omega \in \Omega(\mathcal{H})$ which yields a heuristic with an accurate estimate $h_{\omega}^{SCP}(s)$. Except for very small n , it is obviously impossible to consider all $n!$ orders. Instead, we use hill climbing, a well-known local search technique (Russell and Norvig 1995), to actively *search* in the space of orders.

8.3 Greedy Orders

Before we can start the search, however, we need to address one of the most important questions for local search: where do we start searching? Using a good initial solution is a key ingredient for finding high-quality solutions fast via local search (Lourenço et al. 2010) and many problems allow finding a good initial solution *greedily* (e.g., Korte and Vygen 2001). We use the same approach here and propose an algorithm that starts with an empty order ω and iteratively appends an unordered heuristic to ω until ω contains all heuristics.

But how do we decide which heuristic to append next in each iteration? We could prefer to append heuristics with high estimates for the given state first. This makes it more likely that an accurate heuristic is offered all of the costs it needs for justifying its estimate. However, we also have to keep in mind that usually only the first heuristic is allowed to use all the costs it can exploit. Subsequent heuristics operate on the costs that have not already been consumed by previous heuristics. To preserve costs for as many heuristics as possible, we could let orders begin with the heuristics that “steal” the lowest amount of costs from other heuristics. Finally, we could also prefer heuristics that yield high heuristic estimates *and* steal

few costs. To measure the importance of each objective we introduce three *scoring functions* and we order heuristics by their assigned scores in descending order.

Definition 8.3 Heuristic Scoring Functions.

Let \mathcal{T} be a parameterized-cost transition system and let \mathcal{H} be a set of admissible heuristics for \mathcal{T} , where each $h \in \mathcal{H}$ has a corresponding saturator saturate_h . A scoring function for \mathcal{T} and \mathcal{H} is a function $q : \mathcal{H} \times \mathcal{C}(\mathcal{T}) \times S(\mathcal{T}) \rightarrow \mathbb{R}$.

We define three scoring functions

$$\begin{aligned} q_h(h, \text{cost}, s) &= h(s, \text{cost}), \\ q_{\text{stolen}}(h, \text{cost}, s) &= - \sum_{l \in \mathcal{L}(\mathcal{T}_h)} \text{stolen}(h, \text{cost}, l), \text{ and} \\ q_{\frac{h}{\text{stolen}}}(h, \text{cost}, s) &= \frac{h(s, \text{cost})}{\max(1, \sum_{l \in \mathcal{L}(\mathcal{T}_h)} \text{stolen}(h, \text{cost}, l))} \end{aligned}$$

where

$$\begin{aligned} \text{wanted}(h, \text{cost}, l) &= \text{saturate}_h(\text{cost})(l), \\ \text{free}(h, \text{cost}, l) &= \text{cost}(l) - \sum_{h' \in \mathcal{H} \setminus \{h\}} \text{wanted}(h', \text{cost}, l), \text{ and} \\ \text{stolen}(h, \text{cost}, l) &= \begin{cases} \max(0, \text{wanted}(h, \text{cost}, l) - \text{free}(h, \text{cost}, l)) & \text{if } \text{free}(h, \text{cost}, l) \geq 0 \\ \max(\text{wanted}(h, \text{cost}, l), \text{free}(h, \text{cost}, l)) & \text{otherwise.} \end{cases} \end{aligned}$$

The scoring function q_h assigns high scores to heuristics with high estimates for the given state, while q_{stolen} gives high scores to heuristics stealing few costs from other heuristics. The function $q_{\frac{h}{\text{stolen}}}$ measures how well a heuristic balances the two objectives of having high heuristic value and stealing low costs. We ensure that the divisor is at least 1 to guarantee that the division is always defined.

We now explain the definitions of *wanted*, *free* and *stolen* costs for a heuristic h with saturator saturate_h , a cost function cost and a label l . We say that the saturated costs $\text{saturate}_h(\text{cost})(l)$ form the part of $\text{cost}(l)$ that h *wants*. Then $\text{free}(h, \text{cost}, l)$ are the costs of l that remain for h after giving all other heuristics the costs of l that they want.

The costs of label l that a heuristic h steals from the other heuristics, i.e., $\text{stolen}(h, \text{cost}, l)$, mainly depends on the costs h gets for free. If the amount of free costs is non-negative, h steals the costs it wants minus the costs it gets for free. If h gets more costs for free than it wants, it steals no costs. If h gets no costs for free, the second case applies. When h wants costs ≥ 0 , the stolen costs equal the wanted costs, since $\text{free}(h, \text{cost}, l) \leq 0$. Otherwise,

$cost(l)$	$wanted(h', cost, l)$	$free(h, cost, l)$	$wanted(h, cost, l)$	$stolen(h, cost, l)$
20	5	$20 - 5 = 15$	10	$\max(0, 10 - 15) = 0$
20	15	$20 - 15 = 5$	10	$\max(0, 10 - 5) = 5$
20	5	$20 - 5 = 15$	-2	$\max(0, -2 - 15) = 0$
20	25	$20 - 25 = -5$	10	$\max(10, -5) = 10$
20	25	$20 - 25 = -5$	-2	$\max(-2, -5) = -2$
20	25	$20 - 25 = -5$	-10	$\max(-10, -5) = -5$

Table 8.1: Examples showing how to compute $stolen(h, cost, l)$, i.e., the amount of costs for label l that heuristic h steals from heuristic h' under cost function $cost$.

Algorithm 8 Dynamic greedy ordering algorithm. Given a set of admissible heuristics \mathcal{H} with corresponding saturators, a cost function $cost$, a state s and a scoring function q , it computes a dynamic greedy order by iteratively appending the heuristic with the highest score and updating the estimates and saturated costs for each unordered heuristic.

```

1: function DYNAMICGREEDYORDER( $\mathcal{H}, cost, s, q$ )
2:    $\omega \leftarrow \langle \rangle$ 
3:   while  $\mathcal{H} \neq \emptyset$  do
4:      $h \leftarrow \arg \max_{h' \in \mathcal{H}} q(h', cost, s)$ 
5:      $\omega \leftarrow \omega \oplus \langle h \rangle$ 
6:      $\mathcal{H} \leftarrow \mathcal{H} \setminus \{h\}$ 
7:      $cost \leftarrow cost - saturate_h(cost)$ 
8:   return  $\omega$ 

```

the amount of stolen costs is the maximum over the two negative values of wanted and free costs. This implies that h steals negative costs, i.e., it provides costs for other heuristics that want them. Table 8.1 holds several examples that show how to compute $stolen(h, cost, l)$.

Dynamic Greedy Orders We can plug any of the scoring functions into Algorithm 8 to greedily compute a heuristic order. Given a set of admissible heuristics \mathcal{H} , a cost function $cost$, a state s and a scoring function q , the greedy algorithm starts with an empty order ω and then iteratively appends the heuristic with the highest score under q and updates the remaining cost function $cost$ until all heuristics are part of ω . If there are multiple heuristics with the same score, we break ties arbitrarily.

Comparison of Scoring Functions We evaluate the ordering algorithm and the three scoring functions in a small experiment. For each task in our benchmark set we compute the three different dynamic greedy orders for the initial state and a random order. In Table 8.2, we can see that q_h (maximizing heuristic values) and q_{stolen} (minimizing the sum of stolen

	<i>random</i>	<i>h</i>	<i>stolen</i>	$\frac{h}{stolen}$
<i>random</i>	–	309	391	109
<i>h</i>	772	–	619	262
<i>stolen</i>	662	467	–	78
$\frac{h}{stolen}$	964	601	830	–

Table 8.2: Pairwise comparison of random orders and dynamic greedy orders using different scoring functions. The entry in row r and column c holds the number of tasks in which order r yields a heuristic with a higher heuristic estimate for the initial state than order c . For each comparison we highlight the order with more such tasks in bold.

Algorithm 9 Static greedy ordering algorithm. Given a set of admissible heuristics \mathcal{H} , a cost function $cost$, a state s and a scoring function q , sorts the heuristics by their respective scores in descending order.

- 1: **function** STATICGREEDYORDER($\mathcal{H}, cost, s, q$)
 - 2: **return** $\langle h_1, \dots, h_n \rangle \in \Omega(\mathcal{H})$ with $q(h_i, cost, s) \geq q(h_{i+1}, cost, s)$
-

costs) usually yield better orders than random orders for the initial state. However, there are many tasks where using random orders is preferable to using q_h or q_{stolen} . Between the two functions there is a slight advantage for q_h . However, combining the two functions in $q_{\frac{h}{stolen}}$ leads to higher estimates in the vast majority of tasks compared to all other scoring functions and random orders. Due to these results we only use the scoring function $q_{\frac{h}{stolen}}$ below.

The dynamic ordering algorithm has the drawback that all heuristic estimates and all minimum saturated cost functions have to be recomputed for all remaining heuristics \mathcal{H} in each iteration. For each heuristic this entails running a uniform cost search in the associated abstract transition system, which can take seconds for very large abstractions. Since ordering n heuristics involves running $n(n-1)/2$ uniform cost searches, this quadratic scaling behavior can lead to one algorithm run taking minutes.

Static Greedy Orders By removing line 7 from Algorithm 8 we obtain a *static* version of the ordering algorithm. It precomputes the heuristic estimate and saturated cost function for each heuristic once and always returns a greedy order in less than 0.1 seconds. Since the static greedy ordering algorithm does not iteratively update the cost function, we can rewrite the algorithm without using a while loop as shown in Algorithm 9.

We compare dynamic and static greedy orders empirically by computing both orders for the initial state of each task in our benchmark set. The top left 3×3 subtable in Table 8.3 compares the resulting heuristic values for the initial state between dynamic and static orders. Both greedy orders

	random	dynamic	static	random-opt-1000s	dynamic-opt-1000s	static-opt-1000s
random	-	109	199	0	6	28
dynamic	964	-	451	195	0	40
static	860	134	-	161	7	0
random-opt-1000s	1119	536	714	-	75	109
dynamic-opt-1000s	1069	640	756	315	-	141
static-opt-1000s	1115	602	785	316	105	-

Table 8.3: Pairwise comparison of algorithms ordering heuristics for saturated cost partitioning. The entry in row r and column c holds the number of tasks in which algorithm r yields a heuristic with a higher heuristic estimate for the initial state than algorithm c . For each comparison we highlight the algorithm with more such tasks in bold. The *-opt-1000s orders are optimized with hill climbing for at most 1000 seconds.

outperform random orders by a clear margin, but the dynamic version has an edge over the static one: the former produces a better order than the latter for 451 tasks, while the opposite is the case for only 134 tasks.

As noted above, the higher heuristic values come at a price though. Figure 8.4 shows that static greedy orders are found much faster than dynamic greedy orders for all evaluated tasks, often by a margin of several orders of magnitude. While we can compute a static greedy order in under one millisecond for the majority of tasks, there are many tasks for which we fail to compute a dynamic greedy order in 30 minutes.

Computing orders quickly will be essential in later experiments but we also want orders that result in accurate heuristics. Therefore, the next section considers both dynamic greedy orders, which are slower to compute but result in more accurate heuristics, and static greedy orders, which are faster to compute but yield less accurate heuristics.

8.4 Optimized Orders

When solving an optimization problem, finding a greedy order is often just the first step. To further optimize an order ω for a given state s and cost function $cost$, we propose a hill-climbing search in the space of orders. Starting from the incumbent order ω , we generate neighboring

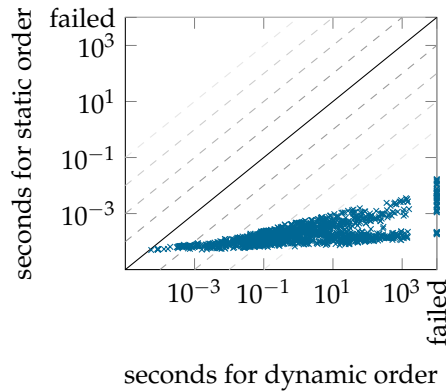


Figure 8.4: Time in seconds for computing a single dynamic greedy and static greedy order (excluding the time for computing a saturated cost partitioning). Each cross corresponds to a task from the benchmark set.

orders by switching any two positions in ω . More precisely, we switch positions 1 and 2, 1 and 3, \dots , 1 and n , 2 and 3, 2 and 4, \dots , 2 and n , etc. This *two-exchange* neighborhood is common for local search optimization algorithms (Pisinger and Ropke 2010) and guarantees that all orders can be reached from any initial order. The first neighbor ω' with $h_{\omega'}^{\text{SCP}}(s, \text{cost}) > h_{\omega}^{\text{SCP}}(s, \text{cost})$ becomes the new incumbent. We repeat this procedure until no neighbor is better than the incumbent or until a time-out is reached.

In addition to this *simple* hill climbing version, we also experimented with *steepest-ascent* hill climbing. The difference between the two versions is that the former commits to the first improving neighbor immediately, while the latter evaluates all neighbors before choosing the best neighbor. The quality of the resulting orders is roughly the same for both hill climbing variants, but steepest-ascent hill climbing usually needs more time to find them. This is not surprising since it has to evaluate $\binom{n}{2} = n(n-1)/2$ neighbors in each iteration, where n is the number of heuristics. Due to this result, we only use simple hill climbing below.

Example Figure 8.5 shows an example run of the hill climbing algorithm optimizing the order of three heuristics h_1 , h_2 and h_3 . In our example the first incumbent order is $\langle h_3, h_2, h_1 \rangle$ with a heuristic value of 5 for the given state s and cost function cost . Its first neighboring order $\langle h_3, h_2, h_1 \rangle$ yields a lower heuristic value, so we turn to the next neighbor $\langle h_1, h_3, h_2 \rangle$. This order yields a higher heuristic value ($h_{\langle h_1, h_3, h_2 \rangle}^{\text{SCP}}(s, \text{cost}) = 7$) than the incumbent order, so we make $\langle h_1, h_3, h_2 \rangle$ the new incumbent³. In the next

³ Notice that this leads to skipping the third neighbor $\langle h_2, h_1, h_3 \rangle$. In contrast, steepest-ascent hill climbing would evaluate all neighbors.

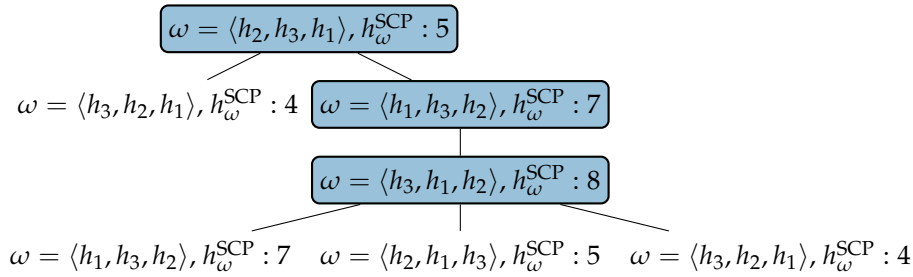


Figure 8.5: Example run of the hill climbing algorithm optimizing the order of three heuristics. We highlight the incumbent order in each iteration of the hill climbing algorithm.

Time limit	0s	1s	5s	10s	50s	100s	500s	1000s	1500s
random-opt	854	883	898	901	925	929	934	937	929
dynamic-opt	929	935	939	939	948	950	941	928	893
static-opt	938	947	959	960	969	972	974	977	969

Table 8.4: Number of solved tasks by saturated cost partitioning using different ordering algorithms and optimization timeouts.

round, the first neighbor $\langle h_3, h_1, h_2 \rangle$ becomes the new incumbent. Afterwards, none of the neighbors improves upon the incumbent, so we abort the procedure and return the incumbent $\langle h_3, h_1, h_2 \rangle$.

Table 8.4 compares the number of solved tasks for random, dynamic greedy and static greedy orders using different optimization time limits. For time limits up to 500 seconds dynamic greedy orders are preferable to random orders. For higher time limits, random orders lead to a higher coverage than dynamic greedy orders. Static greedy orders solve as many or more tasks than dynamic greedy orders and random orders for all evaluated time limits. Random and static greedy orders reach the maximum coverage score when using at most 1000 seconds for hill climbing. For dynamic greedy orders 100 seconds of optimization work best.

As can be expected, random orders benefit from optimization the most. Non-optimized random orders solve 854 tasks, whereas random orders optimized for 1000 seconds solve 937 tasks, a difference of 83 tasks. For dynamic and static greedy orders the maximum difference amounts to 21 and 39 additionally solved tasks, respectively.

In Table 8.3, we compare orders optimized for 1000 seconds to their non-optimized counterparts. Depending on whether we start with a random, dynamic greedy or static greedy order, 1119, 640 and 785 tasks benefit from the optimization. The results also show that optimization almost cancels out the advantage that non-optimized dynamic greedy orders have over non-optimized static greedy ones: optimized dynamic greedy orders are

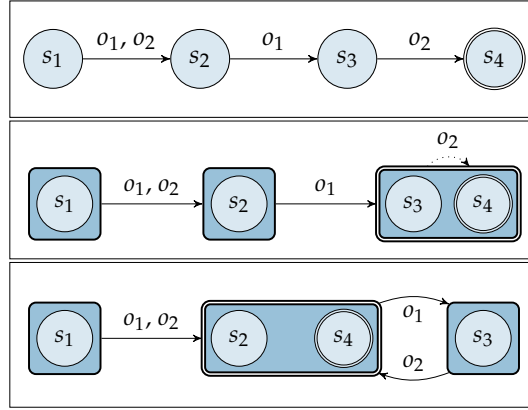


Figure 8.6: Concrete transition system \mathcal{T} (top) with abstraction heuristics h_1 (middle) and h_2 (bottom) used in the proof of Theorem 8.2. All labels have cost 1.

better than optimized static greedy orders in 141 tasks, but the opposite is true in 105 tasks as well.

Optimizing random orders often produces higher estimates compared to using non-optimized dynamic greedy orders (536 vs. 195 tasks). However, optimized dynamic greedy orders are usually preferable to optimized random orders (315 vs. 75 tasks). We can see a similar picture when comparing static greedy orders to random orders. This shows that for obtaining a good order it is beneficial to start with a greedy order and optimize it afterwards.

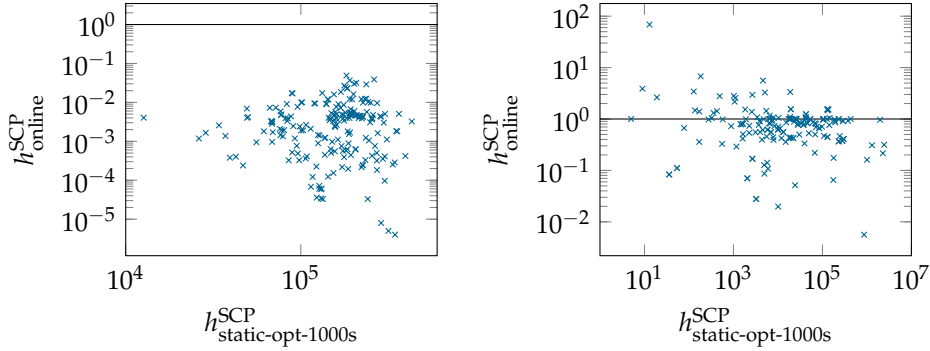
In the experiments below it is even more important to find good orders quickly. Since the computation of dynamic greedy orders takes very long and static greedy orders lead to solving more tasks than dynamic greedy orders, we only use static greedy orders in the following experiments and often refer to them simply as “greedy orders”.

8.5 Online Orders

So far, we have focused on finding an order for a single state. However, as stated above, we need an order that provides good guidance for all states encountered during search. Unfortunately, such an order does not always exist.

Theorem 8.2

There exist regular fixed-cost transition systems $\langle \mathcal{T}, cost \rangle$, sets of cost-monotonic admissible heuristics \mathcal{H} for \mathcal{T} and states $s, s' \in S(\mathcal{T})$ such that $h_\omega^{SCP}(s) > 0$, $h_{\omega'}^{SCP}(s') > 0$, and $h_{\omega''}^{SCP}(s) = h_{\omega'''}^{SCP}(s') = 0$ for two orders $\omega, \omega' \in \Omega(\mathcal{H})$ and all orders $\omega'', \omega''' \in \Omega(\mathcal{H})$ with $\omega'' \neq \omega$ and $\omega''' \neq \omega'$.



(a) Evaluations per second.

(b) Expansions before the last f layer.

Figure 8.7: Comparison of $h_{\text{static-opt-1000s}}^{\text{SCP}}$ and $h_{\text{online}}^{\text{SCP}}$. Each $\langle x, y \rangle$ point corresponds to a task for which $h_{\text{static-opt-1000s}}^{\text{SCP}}$ returns a value of x and $h_{\text{online}}^{\text{SCP}}$ returns $x \cdot y$. Therefore, points below $y = 1$ correspond to tasks where $h_{\text{online}}^{\text{SCP}}$ yields a lower value than $h_{\text{static-opt-1000s}}^{\text{SCP}}$. We exclude tasks for which any of the two algorithms needs less than 1000 evaluations. Note that all axes use a log scale.

Proof. Consider the concrete transition system \mathcal{T} and abstraction heuristics h_1 and h_2 for \mathcal{T} in Figure 8.6. We have $h_{\langle h_1, h_2 \rangle}^{\text{SCP}}(s_2) = 1$, $h_{\langle h_2, h_1 \rangle}^{\text{SCP}}(s_2) = 0$, $h_{\langle h_1, h_2 \rangle}^{\text{SCP}}(s_3) = 0$, and $h_{\langle h_2, h_1 \rangle}^{\text{SCP}}(s_3) = 1$. \square

Theorem 8.2 implies that there are sets of heuristics where no single order yields accurate heuristic estimates for all states. One approach to overcome this problem is to compute a static greedy order and the corresponding saturated cost partitioning in every evaluated state. We hypothesize that this computation is very expensive by itself, so we do not spend additional time to optimize the greedy orders.

The resulting heuristic, called $h_{\text{online}}^{\text{SCP}}$, solves 676 tasks in total, 301 fewer tasks than the best saturated cost partitioning heuristic we have seen so far, $h_{\text{static-opt-1000s}}^{\text{SCP}}$. The difference in coverage stems from the fact that computing a saturated cost partitioning indeed slows down the evaluation significantly. Figure 8.7a compares the number of evaluations per second between $h_{\text{static-opt-1000s}}^{\text{SCP}}$ and $h_{\text{online}}^{\text{SCP}}$. The evaluation is always at least ten times slower for the online version and for many tasks it is more than three orders of magnitude slower. The online version produces somewhat more accurate estimates, as shown in Figure 8.7b, but this is not enough to compensate for the reduced evaluation speed.

Memory, on the other hand, is not a limiting factor for $h_{\text{online}}^{\text{SCP}}$. Even though the algorithm has to hold all abstract transition systems in memory during the search, $h_{\text{online}}^{\text{SCP}}$ never fails to find a plan due to running out of memory.

Our findings for $h_{\text{online}}^{\text{SCP}}$ are in line with other results from the literature which already contains many examples of situations where increased heuristic accuracy does not compensate for the additional computation time spent at every evaluated state (e.g., Karpas et al. 2011; Seipp et al. 2015).

MULTIPLE ORDERS

Since Section 8.5 showed that computing saturated cost partitionings online for every evaluated state is too slow in practice, we pursue an alternative with a good tradeoff between heuristic accuracy and computation time by generating heuristics for *multiple* orders and using the maximum over their estimates in each state.

Generating Multiple Greedy Orders To obtain N greedy orders, we sample N states and compute a greedy order for each of them. We use the sampling procedure by Haslum et al. (2007), using $h_{\text{static}}^{\text{SCP}}$ to estimate the plan cost and to avoid using samples s with $h_{\text{static}}^{\text{SCP}}(s, \text{cost}) = \infty$. A slight modification of the sampling procedure turned out to have a noticeable effect in preliminary experiments: the resulting heuristics were much stronger in a few domains if we ensured that the initial state was part of the samples. This makes sense if we consider that a state-sampling procedure should ideally return states that are similar to the ones expanded during search. Since the initial state is guaranteed to be expanded, it is beneficial to include it in the set of sample states and we do so in all experiments below.

Table 9.1 shows the total coverage scores of saturated cost partitioning heuristics maximizing over N orders optimized for at most X seconds for various values of N and X . We analyze the two dimensions N and X in isolation before looking at their interaction.

Orders	1	2	5	10	20	50	100	200	500
static	938	967	1004	1027	1037	1038	1035	1013	947
static-opt-1s	947	982	1013	1036	1047	1042	1032	1002	919
static-opt-5s	959	978	1017	1039	1053	1044	1033	972	–
static-opt-10s	960	986	1019	1044	1055	1049	1012	324	–
static-opt-50s	969	993	1033	1051	1055	458	–	–	–
static-opt-100s	972	994	1034	1049	544	457	–	–	–
static-opt-500s	974	996	663	–	–	–	–	–	–

Table 9.1: Number of solved tasks when maximizing over saturated cost partitioning heuristics for multiple optimized static greedy orders using different optimization time limits. The static-opt- X s orders are optimized with hill climbing for at most X seconds.

Multiple Orders First, we investigate the impact of changing the number of orders (rows in Table 9.1). The number of solved tasks increases steadily when going from 1 to 50 non-optimized static greedy orders. Afterwards, coverage drops steadily again. The difference between the maximum number of solved tasks and the number of solved tasks by a single order is striking. A single greedy order leads to solving 938 tasks, while using 50 greedy orders leads to solving 1038 tasks, an improvement of 100 tasks. For optimized greedy orders the results are similar: using more than one optimized order is highly beneficial for all tested optimization time limits.

Optimization Next, we look at the influence of optimization on the quality of the resulting heuristics (columns in Table 9.1). The first column repeats the values from the “static-opt” row in Table 8.4. As we saw there, optimizing a single order increases the coverage score. The same result holds when optimizing multiple orders, however the difference in coverage is smaller. This is no surprise, since multiple non-optimized orders already solve many more tasks than single orders. Starting with 20 greedy orders and optimizing them for 10 seconds raises the number of solved tasks from 1037 to 1055. If we optimize for too long, coverage decreases again.

Optimization vs. Multiple Orders Last, we inspect how the number of orders and the optimization time limit interact. As expected, if we use more orders, we need to shorten optimization times. Otherwise, there is not enough time for the A* search. The results also show that using multiple orders is much more important than optimizing them. For example, if we want to use 10 seconds for computing optimized greedy orders, we can optimize 1 order for 10 seconds and solve 960 tasks, 2 orders for 5 seconds and solve 978 tasks, or 10 orders for 1 second and solve 1036 tasks.

Overall, the heuristics with the highest total coverage use 20 optimized static greedy orders. For these heuristics, 10 seconds and 50 seconds of optimization both lead to solving 1055 tasks.

Accuracy vs. Evaluation Time We saw above that using more than 20 greedy orders optimized for 10 seconds leads to fewer solved tasks. Adding an order to an existing set of orders can only increase the accuracy of the resulting heuristic. Since coverage decreases when using more than 20 optimized orders, the gain in accuracy from including additional orders must be outweighed by the increased computational and memory cost. To test this hypothesis, we compare $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ and $h_{100\text{-static-opt-10s}}^{\text{SCP}}$ in Figure 9.1. The left plot (Figure 9.1a) shows the number of evaluations $h_{100\text{-static-opt-10s}}^{\text{SCP}}$ makes per second, relative to the number of evaluations

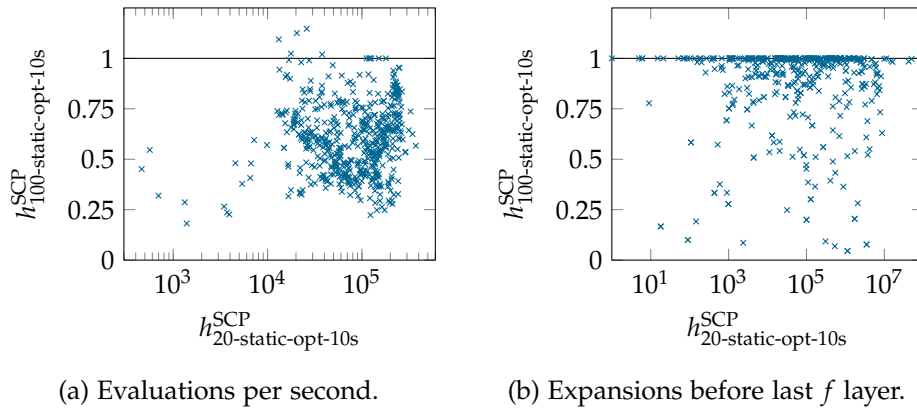


Figure 9.1: Comparison of 20 and 100 saturated cost partitioning heuristics using optimized static greedy orders. Each $\langle x, y \rangle$ point corresponds to a task for which $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ has a value of x and $h_{100\text{-static-opt-10s}}^{\text{SCP}}$ has a value of $x \cdot y$. Therefore, points below $y = 1$ correspond to tasks where $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ has a higher value than $h_{100\text{-static-opt-10s}}^{\text{SCP}}$. We exclude tasks for which any of the two algorithms needs less than 1000 evaluations. Note that the x axis uses a log scale in both plots.

per second by $h_{20\text{-static-opt-10s}}^{\text{SCP}}$. The evaluation speed drops visibly for the vast majority of tasks when adding another 80 optimized greedy orders. For 436 of the 543 commonly solved tasks with at least 1000 evaluations the evaluation speed of $h_{100\text{-static-opt-10s}}^{\text{SCP}}$ drops below 75% of the speed of $h_{20\text{-static-opt-10s}}^{\text{SCP}}$. One might expect that evaluating $h_{100\text{-static-opt-10s}}^{\text{SCP}}$ takes roughly five times as long as evaluating $h_{20\text{-static-opt-10s}}^{\text{SCP}}$, but a significant amount of time is used to look up the abstract states that a given concrete state is mapped to. The time for these computations is independent of the number of orders.

Figure 9.1b reveals that the number of expansions excluding the last f layer remains roughly the same for the majority of tasks and only for 76 of the 543 commonly solved tasks with at least 1000 evaluations the number of expansions decreases by more than 75%. Together, the two plots in Figure 9.1 show that indeed the increase in accuracy does not compensate for the additional evaluation time.

Accuracy vs. Memory Usage In addition to being slower to evaluate, using more orders also results in higher memory usage, since we have to store the abstract goal distances of each abstraction heuristic under each cost partitioning. However, since we use sparse orders and a moderately high memory limit of 3.5 GiB, memory does not affect the coverage score of $h_{100\text{-static-opt-10s}}^{\text{SCP}}$: for all 52 tasks solved by $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ but not by

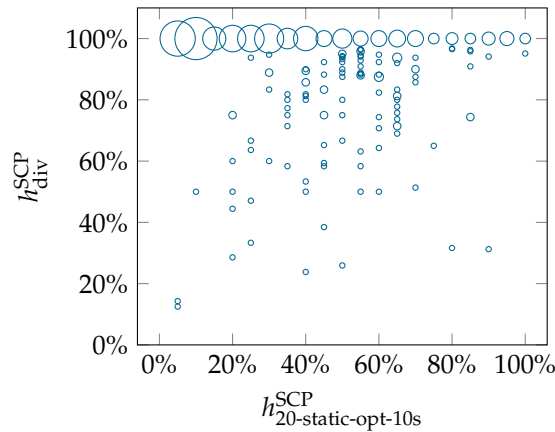


Figure 9.2: Percentage of *probably useful* orders for 20 optimized greedy orders ($h_{20\text{-static-opt-10s}}^{\text{SCP}}$) and diverse orders with a diversification time limit of 200 seconds ($h_{\text{div}}^{\text{SCP}}$). We exclude tasks for which any of the two heuristics uses fewer than 1000 expansions. The area of each circle is proportional to the number of tasks that it represents.

$h_{100\text{-static-opt-10s}}^{\text{SCP}}$, $h_{100\text{-static-opt-10s}}^{\text{SCP}}$ runs out of time and never hits the memory limit.

Probably Useful Orders Our analysis suggests that once the set of orders Ω reaches a certain size, many orders will not contribute to the overall heuristic. To test this hypothesis, we keep track of the sets of orders that induce the highest heuristic estimates for each encountered state. We say that all orders in the *minimal hitting set*¹ of these sets are *useful* for the search, and all others are *useless*. The intuition behind this definition is that a search with a heuristic that discards all heuristics based on useless orders evaluates exactly the same states as a search with all heuristics.

As the computation of a minimum hitting set is NP-complete (Karp 1972), we approximate it by using a greedy algorithm that treats the set of orders Ω as a sequence $\langle \omega_1, \dots, \omega_n \rangle$ (any order suffices). Let $\langle \mathcal{T}, cost \rangle$ be a fixed-cost transition system. Then an order ω_i is *probably useful* for state $s \in S(\mathcal{T})$ if it is the first order in the sequence that maximizes the heuristic value, i.e., $h_{\omega_i}^{\text{SCP}}(s, cost) > h_{\omega_j}^{\text{SCP}}(s, cost)$ for all $j < i$, and $h_{\omega_i}^{\text{SCP}}(s, cost) \geq h_{\omega_j}^{\text{SCP}}(s, cost)$ for all $j > i$. We call all orders that are probably useful for at least one state encountered during search *probably useful*. The set of probably useful orders is a hitting set, but not necessarily a minimal one. It therefore serves as an upper bound on the number of orders that contribute to the search.

¹ We can break ties arbitrarily, so for the sake of simplicity, we assume that there is exactly one minimal hitting set.

Figure 9.2 compares the percentage of probably useful orders for two different heuristics. For the moment, we are only interested in the $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ heuristic on the x -axis, which shows the percentage of probably useful orders out of 20 optimized greedy orders. As we can see, even the strongest saturated cost partitioning heuristic we have described so far, $h_{20\text{-static-opt-10s}'}^{\text{SCP}}$, contains many useless orders. In 35% of the solved tasks at most 20% of the orders are probably useful and in 61% of the solved tasks at most 40% of the orders are probably useful. If we take into account that these numbers are an upper bound on the real number of useful orders, we can confirm that it is rarely useful to add another order to an already sufficiently large set of orders. This raises the question of how to choose a good number of orders, which we focus on next.

9.1 Diverse Orders

The analysis of probably useful orders not only explains why additional orders lead to a lower coverage once Ω reaches a certain size, but it also shows that the convincing results by using multiple greedy orders are obtained *despite* having a large number of useless orders in Ω . Removing the useless orders from Ω would result in faster heuristic evaluation without loss of information, and replacing them with useful orders would result in a more accurate heuristic.

Unfortunately, we can only decide whether an order is useful once the search has terminated. We therefore use the sampling procedure from above to generate a set \hat{S} of 1000 sample states. This sample set serves as a proxy for the real set of states encountered during the search.

Diversification Algorithm We propose the following algorithm for finding a diverse set of useful orders Ω : first, we initialize Ω to be the empty set. Afterwards, until a given time limit T is reached, we iteratively generate a new order ω , add it to Ω if $h_{\omega}^{\text{SCP}}(s, cost) > \max_{\omega' \in \Omega} h_{\omega'}^{\text{SCP}}(s, cost)$ for at least one state $s \in \hat{S}$ and the original cost function $cost$, and discard it otherwise.

This diversification approach has the drawback that it keeps early-found orders with higher probability, even if they are dominated on all sample states by a later-found order. We also experimented with more sophisticated methods but the resulting heuristics were weaker than the ones produced by the diversification procedure above. For example, when we computed a minimal hitting set of orders for the set of samples, the resulting set often contained too few orders. These orders are enough to cover the set of sample states, but other orders which do not have any benefit on the sample set may actually be useful during search.

Diversification time	0s	1s	10s	50s	100s	200s	500s	1000s	1500s
Coverage ₍₁₆₆₇₎	938	1034	1049	1057	1061	1061	1061	1055	1033

Table 9.2: Number of solved tasks by diverse saturated cost partitioning heuristics for static greedy orders using different diversification time limits. The configuration in the left-most column uses a single (non-diverse) order.

	opt-0s	opt-1s	opt-2s	opt-5s	opt-10s	opt-20s	opt-50s	opt-100s	opt-200s	Coverage
opt-0s	–	3	2	5	5	6	8	11	14	1061
opt-1s	2	–	0	4	5	7	9	12	15	1060
opt-2s	3	1	–	4	5	7	9	12	15	1063
opt-5s	2	1	0	–	3	5	6	9	12	1056
opt-10s	3	2	1	2	–	3	6	9	12	1056
opt-20s	3	3	3	4	3	–	5	9	12	1047
opt-50s	2	3	3	4	2	1	–	7	11	1026
opt-100s	2	3	3	3	2	1	0	–	7	1004
opt-200s	2	2	2	2	2	1	0	0	–	979

Table 9.3: Domain-wise coverage comparison of saturated cost partitioning heuristics diversified for 200 seconds using different optimization time limits.

Number of Diverse Orders During the diversification process we do not impose any limit on the number of kept orders and instead let the algorithm find a good size for Ω automatically. In principle, this could lead to using too many orders and therefore slowing down the evaluation too much. However, we hypothesize that if we find another diverse order for the relatively small set of samples, chances are high that it will prove useful during the A^* search as well. We confirmed this hypothesis in preliminary experiments by limiting the number of orders that the diversification method produces. For all tested values of $|\Omega| \geq 20$ the total coverage was almost identical to the number of solved tasks without any size limit on Ω .

We evaluate the diversification procedure using static greedy orders. Table 9.2 shows the total number of tasks solved by the resulting heuristic $h_{\text{div-static}}^{\text{SCP}}$ for various diversification time limits T . It solves more tasks with increasing T until it reaches the peak of 1061 solved tasks at $T = 100$ –500 seconds. Afterwards, coverage decreases again.

We saw in Table 9.1 that the best configuration using non-optimized greedy orders solves 1038 tasks. By diversifying non-optimized greedy orders we are able to raise the total coverage by 23 tasks. In the next experiment, we evaluate whether *optimized* greedy orders also benefit from diver-

sification. We use a fixed diversification time limit of 200 seconds, but vary the time for hill climbing in the space of orders. Figure 9.3 holds coverage results. Two seconds of hill climbing is the setting that yields the strongest heuristic in both a domain-wise and overall coverage comparison. We use the name $h_{\text{div}}^{\text{SCP}}$ for this configuration (200 seconds of diversification and at most 2 seconds of optimization for each greedy order). It solves 1063 of the 1667 tasks in our benchmark set, 8 tasks more than $h_{20\text{-static-opt-10s}}^{\text{SCP}}$, the best configuration using a fixed number of non-diverse optimized greedy orders.

We believe that one of the reasons for $h_{\text{div}}^{\text{SCP}}$ solving more tasks than $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ is that the selected orders are more diverse, and hence there are fewer relevant states where the heuristic guidance of $h_{\text{div}}^{\text{SCP}}$ is poor. This is true even though the average size of Ω is slightly lower for $h_{\text{div}}^{\text{SCP}}$ (arithmetic mean: 13.59 orders), compared to 20 optimized greedy orders. For 271 tasks, only a single order is chosen, and for 11 tasks at least 80 orders are selected during diversification. Even though the percentage of useful orders can be expected to be larger if Ω is smaller, the difference does not make up for the vastly superior impression that can be seen in Figure 9.2: the percentage of probably useful orders of $h_{\text{div}}^{\text{SCP}}$ is higher than for $h_{20\text{-static-opt-10s}}^{\text{SCP}}$ in almost all tasks. Moreover, for 95% of the analyzed tasks more than 60% of the orders of $h_{\text{div}}^{\text{SCP}}$ are probably useful, and there is even a significant amount of tasks (75%) where almost all orders of $h_{\text{div}}^{\text{SCP}}$ (more than 95%) are probably useful.

Related Work Maximizing over multiple saturated cost partitioning heuristics is similar to the approach by Karpas et al. (2011), who maximize over multiple precomputed optimal cost partitioning heuristics. Our method has the advantage that we never have to compute an optimal cost partitioning, which can be prohibitively expensive even for a single computation. Their approach works in practice when using implicit abstractions but it requires too much time and memory for our explicitly represented abstraction heuristics: there are 620 tasks solved by $h_{\text{div}}^{\text{SCP}}$ for which computing even a single optimal cost partitioning is infeasible within 30 minutes and 3.5 GiB. For these tasks, h^{OCP} runs out of time in 359 cases and exceeds the memory limit 261 times.

Choosing a set of diverse saturated cost partitioning heuristics is an instance of the heuristic subset selection problem (e.g., Lelis et al. 2016). It would be interesting to see whether more advanced techniques than our diversification procedure are able to produce stronger overall heuristics.

	$h_{\text{random}}^{\text{SCP}}$	$h_{\text{static}}^{\text{SCP}}$	$h_{\text{static-opt-1000s}}^{\text{SCP}}$	$h_{\text{20-static-opt-10s}}^{\text{SCP}}$	$h_{\text{div}}^{\text{SCP}}$
airport ⁽⁵⁰⁾	25	29	32	30	30
barman ⁽³⁴⁾	4	4	4	4	4
blocks ⁽³⁵⁾	23	28	28	28	28
childsnaack ⁽²⁰⁾	0	0	0	0	0
depot ⁽²²⁾	11	12	12	12	13
driverlog ⁽²⁰⁾	13	14	15	15	15
elevators ⁽⁵⁰⁾	33	37	42	44	44
floortile ⁽⁴⁰⁾	5	5	4	6	6
freecell ⁽⁸⁰⁾	36	61	61	64	65
ged ⁽²⁰⁾	15	15	15	19	19
grid ⁽⁵⁾	3	3	3	3	3
gripper ⁽²⁰⁾	8	8	8	8	8
hiking ⁽²⁰⁾	13	13	13	14	14
logistics ⁽⁶³⁾	25	28	29	36	37
miconic ⁽¹⁵⁰⁾	86	87	87	138	144
movie ⁽³⁰⁾	30	30	30	30	30
mprime ⁽³⁵⁾	27	28	30	31	31
mystery ⁽³⁰⁾	19	18	19	19	19
nomystery ⁽²⁰⁾	16	20	20	20	20
openstacks ⁽¹⁰⁰⁾	50	51	51	51	51
parcprinter ⁽⁵⁰⁾	30	29	39	32	32
parking ⁽⁴⁰⁾	13	13	13	13	13
pathways ⁽³⁰⁾	4	4	4	5	5
pegsol ⁽⁵⁰⁾	44	48	48	48	48
pipes-nt ⁽⁵⁰⁾	21	22	23	24	24
pipes-t ⁽⁵⁰⁾	15	16	17	17	18
psr-small ⁽⁵⁰⁾	50	50	50	50	50
rovers ⁽⁴⁰⁾	8	8	8	8	8
satellite ⁽³⁶⁾	6	7	7	7	7
scanalyzer ⁽⁵⁰⁾	25	23	27	33	33
sokoban ⁽⁵⁰⁾	50	50	50	50	50
storage ⁽³⁰⁾	16	16	16	16	16
tetris ⁽¹⁷⁾	11	11	11	11	11
tidybot ⁽⁴⁰⁾	23	23	23	23	23
tpp ⁽³⁰⁾	7	7	7	8	8
transport ⁽⁷⁰⁾	26	32	32	35	35
trucks ⁽³⁰⁾	9	12	13	13	13
visitall ⁽⁴⁰⁾	13	30	30	30	30
woodwork ⁽⁵⁰⁾	29	34	44	47	45
zenotravel ⁽²⁰⁾	12	12	12	13	13
Sum ⁽¹⁶⁶⁷⁾	854	938	977	1055	1063

Table 9.4: Number of solved tasks by different saturated cost partitioning heuristics. All heuristics use sparse orders.

9.2 Summary of Improvements

In this and the previous chapter, we were able to substantially reduce the memory usage and runtime of saturated cost partitioning heuristics by using sparse orders. On top of that, the preceding experiments have shown four major improvements in quality for heuristics based on saturated cost partitioning: first, by computing a greedy order of heuristics; second, by optimizing the order of heuristics; third, by considering multiple orders; and finally, by explicitly searching for diversity among orders. Table 9.4 shows domain-wise and total coverage results for the heuristics that correspond to these improvements. Using a static greedy order instead of a random one and using 20 optimized static greedy orders instead of a single one led to the biggest changes in total coverage: 84 and 78 additionally solved tasks, respectively. In comparison, optimization and diversification were responsible for smaller differences in coverage and led to solving 39 and 8 additional tasks. All improvements are already impressive by themselves, but even more so, given that each of them is able to raise the total number of solved tasks even after applying the other changes. The strongest heuristic, $h_{\text{div}}^{\text{SCP}}$, is a huge improvement over the saturated cost partitioning heuristic we started with in Chapter 8, $h_{\text{random}}^{\text{SCP}}$: $h_{\text{div}}^{\text{SCP}}$ solves as many or more tasks than $h_{\text{random}}^{\text{SCP}}$ in all domains and raises the total coverage score by 209 tasks.

Part III

COMPARISON OF COST PARTITIONING ALGORITHMS

Even though cost partitioning is an active field of research, apart from a few isolated results, no thorough theoretical or experimental analysis of cost partitioning approaches exists. We provide such an analysis in the remainder of this thesis. First, we show that a key ingredient of saturated cost partitioning — the idea of giving unconsumed costs to other heuristics — can also be applied to uniform cost partitioning, leading to the new *opportunistic* uniform cost partitioning algorithm. Afterwards, we analyze the theoretical relationships between all cost partitioning algorithms presented in this thesis, proving several dominance and non-dominance results. Finally, we compare the algorithms experimentally on pattern databases, Cartesian abstractions and landmark heuristics, showing that saturated cost partitioning is usually the method of choice on the IPC benchmark suite.

THEORETICAL COMPARISON

Uniform cost partitioning suffers from the same problem as greedy zero-one cost partitioning: even if costs are not fully consumed by a heuristic, they are not offered to other heuristics where the increased cost function might lead to higher (yet still admissible) estimates.

10.1 Opportunistic Uniform Cost Partitioning

We propose an opportunistic variant that remedies this shortcoming. Like uniform cost partitioning, we split the label costs evenly among the heuristics affected by a label, but like saturated cost partitioning, we consider heuristics in sequence, let saturators determine the needed costs and redistribute unneeded costs to the heuristics encountered later in the sequence.

Definition 10.1 Opportunistic uniform cost partitioning.

Let $\langle \mathcal{T}, cost \rangle$ be a regular fixed-cost transition system, let \mathcal{H} be a set of admissible heuristics for \mathcal{T} , let $\langle h_1, \dots, h_n \rangle \in \Omega(\mathcal{H})$ be an order of \mathcal{H} and let $\langle saturate_1, \dots, saturate_n \rangle$ be corresponding saturators. The opportunistic uniform cost partitioning $\mathcal{C} = \langle cost_1, \dots, cost_n \rangle$, the remaining cost functions $\langle remain_0, \dots, remain_n \rangle$ and the offered cost functions $\langle \widetilde{cost}_1, \dots, \widetilde{cost}_n \rangle$ are defined by

$$\begin{aligned}
 remain_0 &= cost \\
 \widetilde{cost}_i(l) &= \begin{cases} \frac{remain_{i-1}(l)}{|\{h \in \{h_i, \dots, h_n\} \mid l \in \mathcal{A}(h)\}|} & \text{if } l \in \mathcal{A}(h_i) \\ 0 & \text{otherwise} \end{cases} \quad \text{for all } l \in \mathcal{L}(\mathcal{T}) \\
 cost_i &= saturate_i(\widetilde{cost}_i) \\
 remain_i &= remain_{i-1} - cost_i
 \end{aligned}$$

We write h_ω^{OUCP} for the heuristic that is cost-partitioned by opportunistic uniform cost partitioning for order ω .

Example Consider again the two abstraction heuristics h_1 and h_2 and the original cost function $cost = \langle 4, 1, 4, 1 \rangle$ from Figure 3.1 on p. 15. To compute an opportunistic uniform cost partitioning, we need to order the heuristics. If we use the order $\langle h_1, h_2 \rangle$, h_1 is offered $\widetilde{cost}_1 = \langle 2, 0, 2, 1 \rangle$, the same as when using uniform cost partitioning. The minimum saturated cost function $saturate_{h_1}(\widetilde{cost}_1) = \langle 2, 0, 1, 1 \rangle$ tells us that we can reduce the cost of

operator o_3 from 2 to 1 without affecting any heuristic values of h_1 . Consequently, we offer $\widetilde{cost}_2 = \langle 2, 1, 3, 0 \rangle$ to h_2 . Again, the heuristic does not need all of the offered costs for its estimates: $saturate_{h_2}(\widetilde{cost}_2) = \langle 1, 1, 3, 0 \rangle$. Under the two saturated cost functions we have $h_{\langle h_1, h_2 \rangle}^{OUCP}(s_1, cost) = 3 + 4 = 7$. The heuristic value for s_1 remains the same if we change the order of the two heuristics. However, we have $h_{\langle h_1, h_2 \rangle}^{OUCP}(s_3, cost) = 4$ and $h_{\langle h_2, h_1 \rangle}^{OUCP}(s_3, cost) = 3$.

10.2 Dominances and Non-dominances

We begin with two theorems showing that saving unused costs is beneficial with cost-monotonic heuristics.

Theorem 10.1 $h^{SCP} \geq h^{GZOCP}$.

Let $\langle \mathcal{T}, cost \rangle$ be a regular fixed-cost transition system and let \mathcal{H} be a set of cost-monotonic admissible heuristics for \mathcal{T} . Then $h_{\omega}^{SCP}(s, cost) \geq h_{\omega}^{GZOCP}(s, cost)$ for all orders $\omega \in \Omega(\mathcal{H})$ and all $s \in S(\mathcal{T})$. Moreover, there are cases where the inequality is strict for some $s \in S(\mathcal{T})$ and all orders $\omega \in \Omega(\mathcal{H})$.

Proof. For the second part, the abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 3.1 on p. 15 provide an example with $h_{\omega}^{SCP}(s_1, cost) = 8$ and $h_{\omega}^{GZOCP}(s_1, cost) = 5$ for all $\omega \in \Omega(\mathcal{H})$.

For the first part, we show that the stronger result $h_{\omega'}^{SCP}(s, cost') \geq h_{\omega'}^{GZOCP}(s, cost'')$ holds for all states $s \in S(\mathcal{T})$, all orders of admissible cost-monotonic heuristics ω' and non-negative cost functions $cost', cost'' \in \mathcal{C}(\mathcal{T})$ with $cost' \geq cost''$. The theorem follows from the case $\omega = \omega'$ and $cost = cost' = cost''$.

Strengthening the claim allows proving the result by induction over the length of ω' . For the empty sequence ω' , both heuristics are 0, so the inequality holds trivially.

Otherwise decompose ω' into the first component h_1 and remaining sequence ω'' . The value contributed by h_1 to $h_{\omega'}^{SCP}(s, cost')$ is $h_1(s, cost')$ by definition of saturated cost. The value contributed by h_1 to $h_{\omega'}^{GZOCP}(s, cost'')$ is $h_1(s, cost'')$ because h_1 receives the full operator costs from $cost''$ for all labels affecting h_1 . We get $h_1(s, cost') \geq h_1(s, cost'')$ because h_1 is cost-monotonic.

For labels that do not affect h_1 , both algorithms assign cost 0 to h_1 , and hence the remaining costs for ω'' are at least as large under saturated cost partitioning as under greedy zero-one cost partitioning. For labels that affect h_1 , greedy zero-one cost partitioning uses up the whole cost for h_1 , so the remaining costs for ω'' are again at least as large under saturated cost partitioning as under greedy zero-one cost partitioning because the latter are 0. By the induction hypothesis, the heuristic value contributed by ω' is then at least as large for saturated cost partitioning as for greedy zero-one cost partitioning, concluding the proof. \square

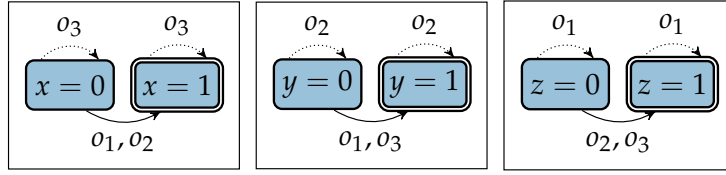


Figure 10.1: Abstraction heuristics used in the proof of Theorem 10.3. The concrete initial state s_0 is $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$. The cost function is $cost = \langle 1, 1, 1 \rangle$, i.e., all operators cost 1.

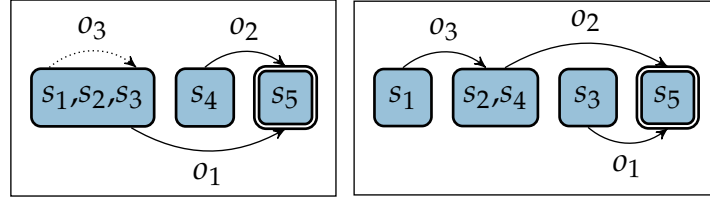


Figure 10.2: Abstraction heuristics used in the proofs of Theorems 10.3 and 10.5. The cost function is $cost = \langle 4, 1, 1 \rangle$, i.e., operator o_1 costs 4 while o_2 and o_3 cost 1.

Theorem 10.2 $h^{\text{OUCP}} \geq h^{\text{UCP}}$.

Let $\langle \mathcal{T}, cost \rangle$ be a regular fixed-cost transition system and let \mathcal{H} be a set of cost-monotonic admissible heuristics for \mathcal{T} . Then $h_\omega^{\text{OUCP}}(s, cost) \geq h^{\text{UCP}}(s, cost)$ for all orders $\omega \in \Omega(\mathcal{H})$ and all $s \in S(\mathcal{T})$. Moreover, there are cases where the inequality is strict for some $s \in S$ and all orders $\omega \in \Omega(\mathcal{H})$.

Proof. For the second part, the abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 3.1 on p. 15 provide an example with $h_\omega^{\text{OUCP}}(s_1, cost) = 7$ for all $\omega \in \Omega(\mathcal{H})$ and $h^{\text{UCP}}(s_1, cost) = 6$.

The proof of the first part is analogous to the proof of Theorem 10.1. The only difference is that only a fraction of the label cost of a label l affecting h_1 may be used for h_1 , but because this fraction is the same for both uniform cost partitioning variants ($1/k$, where k is the number of heuristics in ω' affected by l), this does not make a difference to the proof argument. \square

We now know that there are three cost partitioning algorithms – saturated cost partitioning, opportunistic uniform cost partitioning and post-hoc optimization – that dominate one of the other three discussed algorithms. Next, we show that none of these three dominating algorithms dominates any of the other two.

Theorem 10.3 Comparison of h^{SCP} , h^{OUCP} , and h^{PhO} .

For each of the following pairs of cost partitioning algorithms, there exists a regu-

lar fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ and a set of admissible heuristics \mathcal{H} for \mathcal{T} such that

$$h_{\omega}^{OUCP}(s, cost) > h_{\omega}^{SCP}(s, cost) \quad (10.1)$$

$$h_{\omega}^{SCP}(s, cost) > h_{\omega}^{OUCP}(s, cost) \quad (10.2)$$

$$h^{PhO}(s, cost) > h_{\omega}^{SCP}(s, cost) \quad (10.3)$$

$$h_{\omega}^{SCP}(s, cost) > h^{PhO}(s, cost) \quad (10.4)$$

$$h^{PhO}(s, cost) > h_{\omega}^{OUCP}(s, cost) \quad (10.5)$$

$$h_{\omega}^{OUCP}(s, cost) > h^{PhO}(s, cost) \quad (10.6)$$

for a state $s \in S(\mathcal{T})$ and all orders $\omega \in \Omega(\mathcal{H})$.

Proof. Consider the two abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 3.1 on p. 15. For all orders $\omega \in \Omega(\mathcal{H})$, we have $h_{\omega}^{SCP}(s_1, cost) = 8$, $h_{\omega}^{OUCP}(s_1, cost) = 7$ and $h^{PhO}(s_1, cost) = 5$, showing 10.2, 10.4 and 10.6.

Consider the three abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 10.1. For all orders $\omega \in \Omega(\mathcal{H})$, we have $h_{\omega}^{OUCP}(s_0, cost) = h^{PhO}(s_0, cost) = 0.5 + 0.5 + 0.5 = 1.5$ and $h_{\omega}^{SCP}(s_0, cost) = 1$, showing 10.1 and 10.3.

Consider the two abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 10.2. We have $h^{PhO}(s_1, cost) = 4$ and for all orders $\omega \in \Omega(\mathcal{H})$, $h_{\omega}^{OUCP}(s_1, cost) = 3.5$, showing 10.5. \square

Next, we analyze the relationship of greedy zero-one cost partitioning to other cost partitioning algorithms. We already know that it is dominated by saturated cost partitioning. Now, we show that there is also a connection to the canonical heuristic and its maximal independent subsets.

Theorem 10.4 $h^{GZOCP} \geq \text{Maximal Independent Subset}$.

Let $\langle \mathcal{T}, cost \rangle$ be a regular fixed-cost transition system, let \mathcal{H} be a set of admissible heuristics for \mathcal{T} and let $\sigma \in \text{MIS}(\mathcal{H})$ be a maximal independent subset of \mathcal{H} . Then there is an order $\omega \in \Omega(\mathcal{H})$ with $h_{\omega}^{GZOCP}(s, cost) \geq \sum_{h \in \sigma} h(s, cost)$ for all states $s \in S(\mathcal{T})$. Moreover, there are cases where the inequality is strict for some state $s' \in S(\mathcal{T})$.

Proof. For a maximal independent subset $\sigma \in \text{MIS}(\mathcal{H})$, let $\omega_{\sigma} \in \Omega(\sigma)$ be any order of σ . Furthermore, let $\omega \in \Omega(\mathcal{H})$ be the concatenation of ω_{σ} and an arbitrary order of the remaining heuristics $\mathcal{H} \setminus \sigma$.

Due to the pairwise independence of all heuristics in σ and therefore in ω_{σ} , each label $l \in \mathcal{L}(\mathcal{T})$ affects at most one heuristic in ω_{σ} . Consequently, the greedy zero-one cost partitioning for ω_{σ} gives the full cost of all labels that affect a heuristic $h \in \omega_{\sigma}$ to h and we have $h_{\omega_{\sigma}}^{GZOCP}(s, cost) = \sum_{h \in \omega_{\sigma}} h(s, cost)$ for all states $s \in S(\mathcal{T})$.

Since we can assume that all heuristics in \mathcal{H} only return non-negative values (any negative heuristic value for a regular transition system can be increased to 0 without sacrificing admissibility), appending heuristics to ω_σ can only increase the resulting heuristic value and we have $h_\omega^{\text{GZOCP}}(s, \text{cost}) \geq h_{\omega_\sigma}^{\text{GZOCP}}(s, \text{cost})$ for all states $s \in S(\mathcal{T})$.

As a result, $h_\omega^{\text{GZOCP}}(s, \text{cost}) \geq h_{\omega_\sigma}^{\text{GZOCP}}(s, \text{cost}) = \sum_{h \in \omega_\sigma} h(s, \text{cost}) = \sum_{h \in \sigma} h(s, \text{cost})$ for all states $s \in S(\mathcal{T})$, which proves the claim.

For a case where the inequality is strict, consider the example from Figure 10.2. The two heuristics h_1 and h_2 are not independent, and therefore the maximal independent subsets of \mathcal{H} are $\{h_1\}$ and $\{h_2\}$. We have $h_1(s_1, \text{cost}) = 4$ and $h_{\langle h_1, h_2 \rangle}^{\text{GZOCP}}(s_1, \text{cost}) = 5$. \square

If we construct a suitable order for each maximal independent subset, as described in the proof of Theorem 10.4, and maximize over the greedy zero-one cost partitionings computed for these orders in the same way that h^{CAN} maximizes over independent subsets of heuristics, the resulting heuristic dominates h^{CAN} .

Theorem 10.5 Maximum Over h^{GZOCP} Heuristics $\geq h^{\text{CAN}}$.

Let $\langle \mathcal{T}, \text{cost} \rangle$ be a regular fixed-cost transition system and let \mathcal{H} be a set of admissible heuristics for \mathcal{T} . Then there is a set of orders $\Omega \subseteq \Omega(\mathcal{H})$ with $\max_{\omega \in \Omega} h_\omega^{\text{GZOCP}}(s, \text{cost}) \geq h^{\text{CAN}}(s, \text{cost})$ for all states $s \in S(\mathcal{T})$. Moreover, there are cases where the inequality is strict for some state $s' \in S(\mathcal{T})$.

Proof. We start with an empty set of orders Ω . Then for each $\sigma \in \text{MIS}(\mathcal{H})$, we construct an order ω as outlined in the proof of Theorem 10.4 and add it to Ω . For each state $s \in S(\mathcal{T})$ there is a maximal independent subset $\sigma \in \text{MIS}(\mathcal{H})$ that has the highest heuristic value $\sum_{h \in \sigma} h(s, \text{cost})$ for s among all maximal independent subsets. The claim follows from the fact that Ω contains an order ω for which $h_\omega^{\text{GZOCP}}(s, \text{cost}) \geq \sum_{h \in \sigma} h(s, \text{cost})$.

For a case where the inequality is strict, consider the example from Figure 10.2. The two heuristics h_1 and h_2 are not independent, and therefore the set of all maximal independent subsets of \mathcal{H} contains each heuristic individually, yielding $h^{\text{CAN}}(s_1, \text{cost}) = \max(4, 2) = 4$. However, $\max(h_{\langle h_1, h_2 \rangle}^{\text{GZOCP}}(s_1, \text{cost}), h_{\langle h_2, h_1 \rangle}^{\text{GZOCP}}(s_1, \text{cost})) = \max(5, 2) = 5$. \square

Corollary 10.1 Maximum Over h^{SCP} Heuristics $\geq h^{\text{CAN}}$.

Let $\langle \mathcal{T}, \text{cost} \rangle$ be a regular fixed-cost transition system and let \mathcal{H} be a set of admissible heuristics for \mathcal{T} . Then there is a set of orders $\Omega \subseteq \Omega(\mathcal{H})$ where $\max_{\omega \in \Omega} h_\omega^{\text{SCP}}(s, \text{cost}) \geq h^{\text{CAN}}(s, \text{cost})$ for all states $s \in S(\mathcal{T})$. Moreover, there are cases where the inequality is strict for some state $s' \in S(\mathcal{T})$.

Proof. Follows directly from Theorems 10.1 and 10.5. \square

We remark that the canonical heuristic still has an advantage over h^{GZOCP} and h^{SCP} , namely that it suffices to compute the component heuristics w.r.t. a single cost function. This is different for cost partitioning algorithms that maximize over a (possibly large) number of orders, with each order requiring a different cost function for each heuristic. This can be a concern especially for memory-based heuristics like PDBs, where each cost function requires a separate PDB.

Greedy Zero-One Cost Partitioning The quality of heuristics based on greedy zero-one cost partitioning strongly depends on the order in which the component heuristics are considered. Consequently, we can always find suitable heuristics and order them in a way so that h^{GZOCP} yields either a lower or a higher estimate than h^{UCP} , h^{OUCP} , and h^{PhO} .

Theorem 10.6 Comparison of h^{GZOCP} to h^{UCP} , h^{OUCP} , and h^{PhO} .

For each of the following pairs of cost partitioning algorithms, there exists a regular fixed-cost transition system $\langle \mathcal{T}, \text{cost} \rangle$ and a set of admissible heuristics \mathcal{H} for \mathcal{T} such that

$$\begin{aligned} h^{\text{GZOCP}}(s, \text{cost}) &> h^{\text{UCP}}(s, \text{cost}) \\ h^{\text{UCP}}(s, \text{cost}) &> h_{\omega}^{\text{GZOCP}}(s, \text{cost}) \\ h_{\omega}^{\text{GZOCP}}(s, \text{cost}) &> h_{\omega}^{\text{OUCP}}(s, \text{cost}) \\ h_{\omega}^{\text{OUCP}}(s, \text{cost}) &> h_{\omega}^{\text{GZOCP}}(s, \text{cost}) \\ h^{\text{GZOCP}}(s, \text{cost}) &> h^{\text{PhO}}(s, \text{cost}) \\ h^{\text{PhO}}(s, \text{cost}) &> h_{\omega}^{\text{GZOCP}}(s, \text{cost}) \end{aligned}$$

for a state $s \in S(\mathcal{T})$ and at least one order $\omega \in \Omega(\mathcal{H})$.

Proof. Consider the two heuristics h_1 and h_2 and cost function cost in Figure 10.2. We have $h_{\langle h_2, h_1 \rangle}^{\text{GZOCP}}(s_1, \text{cost}) = 2$, $h^{\text{UCP}}(s_1, \text{cost}) = h_{\langle h_1, h_2 \rangle}^{\text{OUCP}}(s_1, \text{cost}) = h_{\langle h_2, h_1 \rangle}^{\text{OUCP}}(s_1, \text{cost}) = 3.5$, $h^{\text{PhO}}(s_1, \text{cost}) = 4$ and $h_{\langle h_1, h_2 \rangle}^{\text{GZOCP}}(s_1, \text{cost}) = 5$. \square

In our final analysis, we investigate the remaining relationships between (opportunistic) uniform cost partitioning, the canonical heuristic and post-hoc optimization.

Theorem 10.7 Remaining Comparisons.

For each of the following pairs of cost partitioning algorithms, there exists a regu-

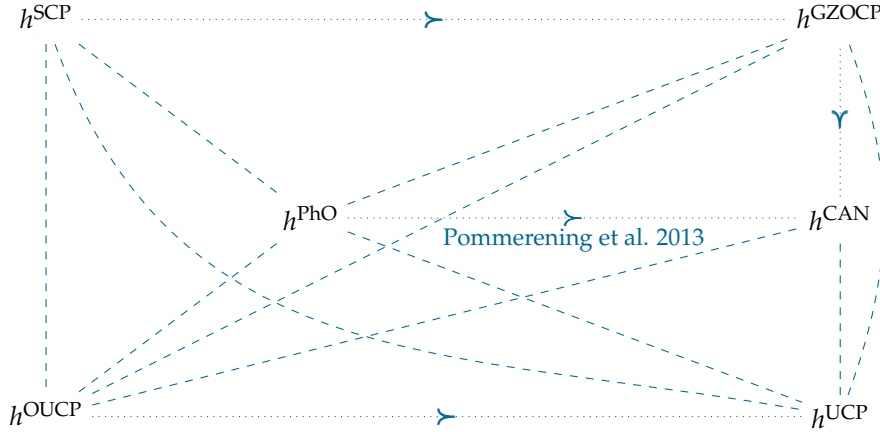


Figure 10.3: Theoretical relationships between the different cost partitioning algorithms. Successor signs stand for dominances, dashed lines indicate incomparable algorithm pairs.

lar fixed-cost transition system $\langle \mathcal{T}, cost \rangle$ and a set of admissible heuristics \mathcal{H} for \mathcal{T} such that

$$h^{UCP}(s, cost) > h^{PhO}(s, cost) \quad (10.7)$$

$$h^{PhO}(s, cost) > h^{UCP}(s, cost) \quad (10.8)$$

$$h^{UCP}(s, cost) > h^{CAN}(s, cost) \quad (10.9)$$

$$h^{CAN}(s, cost) > h^{UCP}(s, cost) \quad (10.10)$$

$$h_{\omega}^{OUCP}(s, cost) > h^{CAN}(s, cost) \quad (10.11)$$

$$h^{CAN}(s, cost) > h_{\omega}^{OUCP}(s, cost) \quad (10.12)$$

for a state $s \in S(\mathcal{T})$ and all orders $\omega \in \Omega(\mathcal{H})$.

Proof. Consider the abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 3.1 on p. 15. For all orders $\omega \in \Omega(\mathcal{H})$, we have $h_{\omega}^{OUCP}(s_1, cost) = 7$, $h^{UCP}(s_1, cost) = 6$, $h^{PhO}(s_1, cost) = 5$ and $h^{CAN}(s_1, cost) = 5$, showing 10.7, 10.9 and 10.11.

Consider the abstraction heuristics \mathcal{H} and cost function $cost$ in Figure 10.2. For all orders $\omega \in \Omega(\mathcal{H})$, we have $h_{\omega}^{OUCP}(s_1, cost) = 3.5$, $h^{UCP}(s_1, cost) = 3.5$, $h^{PhO}(s_1, cost) = 4$ and $h^{CAN}(s_1, cost) = 4$, showing 10.8, 10.10 and 10.12. \square

Figure 10.3 illustrates the theoretical relationships between the different cost partitioning algorithms. This concludes our theoretical investigation, and we now turn to the experimental analysis.

EXPERIMENTAL COMPARISON

We showed in Chapter 8 that the order in which saturated cost partitioning considers the component heuristics greatly influences the quality of the resulting heuristic. The choice of order is also relevant for opportunistic uniform cost partitioning and greedy zero-one cost partitioning. Therefore, we need to decide how to order the component heuristics before we can evaluate these order-dependent algorithms.

11.1 Heuristic Orders

For saturated cost partitioning we obtained the best orders by using a greedy algorithm for finding an initial order and then optimizing it with a hill-climbing search in the space of orders. Before we look into optimization via hill climbing, we analyze whether opportunistic uniform cost partitioning and greedy-zero one cost partitioning also benefit from using greedy orders instead of random ones.

We quickly remind the reader of the three greedy ordering variants:

- *h*: Sort heuristics in descending order by their heuristic value for the given state.
- *stolen*: Sort heuristics in ascending order by the amount of costs they steal from other heuristics.
- $\frac{h}{stolen}$: Sort heuristics in descending order by their heuristic estimate divided by the amount of costs they steal from other heuristics.

We showed in Section 8.3 that $\frac{h}{stolen}$ yields the best greedy orders for saturated cost partitioning. To find the best greedy order for opportunistic uniform cost partitioning we run a small experiment. For each task in our benchmark set with initial state s_0 and cost function $cost$ we compute a random and the three greedy orders ω for s_0 and compare the resulting heuristic estimates $h_{\omega}^{OUCP}(s_0, cost)$. As in Chapters 8 and 9, we use the combined set of abstraction heuristics (COMB) for evaluating orders, i.e., the union of hill climbing pattern databases, systematic pattern databases and Cartesian abstraction heuristics.

Table 11.1a compares the resulting heuristic values. For each pair of orders ω and ω' , the table shows the number of tasks for which ω yields a higher heuristic than ω' for the initial state and vice versa. We can see that

	<i>random</i>	<i>h</i>	<i>stolen</i>	$\frac{h}{stolen}$		<i>random</i>	<i>h</i>	<i>stolen</i>	$\frac{h}{stolen}$
<i>random</i>	–	1248	358	1055	<i>random</i>	–	135	846	552
<i>h</i>	99	–	176	381	<i>h</i>	1321	–	1388	1158
<i>stolen</i>	834	1239	–	1144	<i>stolen</i>	535	194	–	223
$\frac{h}{stolen}$	193	694	34	–	$\frac{h}{stolen}$	737	217	833	–

(a) Opportunistic uniform cost partitioning. (b) Greedy zero-one cost partitioning.

Table 11.1: Pairwise comparison of random orders and greedy orders. The entry in row r and column c holds the number of tasks in which order r yields a heuristic with a higher heuristic estimate for the initial state than order c . For each comparison we highlight the order with more such tasks in bold.

preferring heuristics with high estimates (h) usually yields orders with lower estimates than random orders. To a lesser extent, the same is true for the greedy variant that computes the ratio of heuristic estimate divided by the sum of stolen costs ($\frac{h}{stolen}$). The only greedy order that outperforms random orders is *stolen*. We believe that *stolen* achieves the best results for h^{OUCP} because it makes h^{OUCP} distribute costs conservatively, while the other orders may often lead to high cost being used too early. In other words, *stolen* puts those heuristics in front for which lots of labels have low saturated cost. Consequently, these labels can be assigned higher costs in the heuristics that can actually use high costs.

For greedy zero-one cost partitioning the results are different. Table 11.1b shows that basing the ordering on heuristic values is crucial in this setting. Both h and $\frac{h}{stolen}$ outperform *stolen*. The greedy algorithm produces the best orders if it only takes into account the heuristic estimates (h). This is to be expected, as greedy zero-one cost partitioning does not reuse any unused costs. Consequently, there is no reason for minimizing stolen costs.

In summary, each of the three greedy orders is preferable for one of the three order-dependent cost partitioning algorithms. For opportunistic uniform cost partitioning it is best to prefer heuristics that steal few costs, whereas for greedy zero-one cost partitioning one should prefer heuristics with high estimates. For saturated cost partitioning we obtain the best orders by combining these two objectives.

Now that we know how to find a good initial order for all order-dependent cost partitioning algorithms, we look into optimizing the initial order. For saturated cost partitioning we saw that a limit of 1000 seconds for optimization works best. In Tables 11.2a and 11.2b we find suitable limits for h^{OUCP} (500 seconds) and h^{GZOC} (1000 seconds).

In the experiments below, we find a greedy order for each type of cost partitioning with its preferred greedy variant G and optimize the order

	$h_{\text{one-opt-100s}}^{\text{OUCP}}$	$h_{\text{one-opt-500s}}^{\text{OUCP}}$	$h_{\text{one-opt-1000s}}^{\text{OUCP}}$	$h_{\text{one-opt-1500s}}^{\text{OUCP}}$		$h_{\text{one-opt-100s}}^{\text{GZOCP}}$	$h_{\text{one-opt-500s}}^{\text{GZOCP}}$	$h_{\text{one-opt-1000s}}^{\text{GZOCP}}$	$h_{\text{one-opt-1500s}}^{\text{GZOCP}}$
Coverage <small>(1667)</small>	813	814	798	763	Coverage <small>(1667)</small>	796	797	799	793

(a) Opportunistic uniform cost partitioning. (b) Greedy zero-one cost partitioning.

Table 11.2: Number of solved tasks by single-order cost partitioning heuristics using different optimization time limits.

via hill climbing for its preferred optimization time limit X . We denote the resulting heuristics by $h_{\text{one}}^{\text{OUCP}}$ ($G = \textit{stolen}$, $X = 500$ seconds), $h_{\text{one}}^{\text{GZOCP}}$ ($G = h$, $X = 1000$ seconds) and $h_{\text{one}}^{\text{SCP}}$ ($G = \frac{h}{\textit{stolen}}$, $X = 1000$ seconds).

Multiple Orders In Chapter 9, we showed for saturated cost partitioning that using multiple orders and maximizing over the produced cost partitioning heuristics significantly outperforms single-order cost-partitioned heuristics. In light of this result, the experiments below also consider versions of h^{OUCP} , h^{GZOCP} and h^{SCP} that maximize over multiple cost-partitioned heuristics. We use the diversification procedure from Section 9.1 to obtain a diverse set of cost-partitioned heuristics computed for optimized greedy orders and denote the resulting heuristics by $h_{\text{div}}^{\text{OUCP}}$, $h_{\text{div}}^{\text{GZOCP}}$ and $h_{\text{div}}^{\text{SCP}}$.

For $h_{\text{div}}^{\text{SCP}}$ the best parameter setting uses 200 seconds of diversification and optimizes each greedy order for at most 2 seconds (see Section 9.1). We use the same diversification time limit of 200 seconds in our experiments for $h_{\text{div}}^{\text{OUCP}}$ and $h_{\text{div}}^{\text{GZOCP}}$. To find the best optimization time limits for the two heuristics, we compare different values in Tables 11.3a and 11.3b. The domain-wise and total coverage scores show that for both $h_{\text{div}}^{\text{OUCP}}$ and $h_{\text{div}}^{\text{GZOCP}}$ optimizing for 50 seconds works best, so we use this setting in the experiments below.

Having decided how to compute orders for all order-dependent cost partitioning algorithms, we are now ready to compare all cost partitioning algorithms experimentally.

11.2 Abstraction Heuristics

We begin by computing cost partitionings for four different sets of abstraction heuristics (see Chapter 8 for a description of the heuristics):

- PDBs found by hill climbing (HC)
- PDBs for systematically generated patterns of sizes 1 and 2 (Sys)

	$h_{\text{div-opt-1s}}^{\text{OUCP}}$	$h_{\text{div-opt-5s}}^{\text{OUCP}}$	$h_{\text{div-opt-10s}}^{\text{OUCP}}$	$h_{\text{div-opt-50s}}^{\text{OUCP}}$	$h_{\text{div-opt-100s}}^{\text{OUCP}}$	Coverage		$h_{\text{div-opt-1s}}^{\text{GZOCP}}$	$h_{\text{div-opt-5s}}^{\text{GZOCP}}$	$h_{\text{div-opt-10s}}^{\text{GZOCP}}$	$h_{\text{div-opt-50s}}^{\text{GZOCP}}$	$h_{\text{div-opt-100s}}^{\text{GZOCP}}$	Coverage
$h_{\text{div-opt-1s}}^{\text{OUCP}}$	–	1	2	0	1	806	$h_{\text{div-opt-1s}}^{\text{GZOCP}}$	–	2	3	3	8	823
$h_{\text{div-opt-5s}}^{\text{OUCP}}$	2	–	2	0	1	806	$h_{\text{div-opt-5s}}^{\text{GZOCP}}$	0	–	1	1	6	821
$h_{\text{div-opt-10s}}^{\text{OUCP}}$	2	1	–	0	1	806	$h_{\text{div-opt-10s}}^{\text{GZOCP}}$	2	2	–	1	6	821
$h_{\text{div-opt-50s}}^{\text{OUCP}}$	4	3	4	–	2	812	$h_{\text{div-opt-50s}}^{\text{GZOCP}}$	5	5	4	–	8	825
$h_{\text{div-opt-100s}}^{\text{OUCP}}$	3	2	3	0	–	810	$h_{\text{div-opt-100s}}^{\text{GZOCP}}$	3	3	3	1	–	814

(a) Opportunistic uniform cost partitioning. (b) Greedy zero-one cost partitioning.

Table 11.3: Domain-wise and total coverage comparison of diverse cost partitioning heuristics using different optimization time limits and a fixed diversification time limit of 200 seconds. Both tables consist of a left and right subtable. Left: Pairwise comparison. The entry in row r and column c holds the number of domains in which algorithm r solved more tasks than algorithm c . For each algorithm pair we highlight the maximum of the entries (r, c) and (c, r) in bold. Right: Total number of solved tasks by each algorithm.

- Cartesian abstraction heuristics for the landmark and goal task decompositions (CART)
- Combined abstraction heuristics (COMB), i.e., $\text{COMB} = \text{HC} \cup \text{Sys} \cup \text{CART}$.

We list detailed results for the four settings in the appendix (Tables A.1, A.2, A.3 and A.4). Table 11.4 shows an overview of the results across all settings. We discuss the questions and results in the table from top to bottom.

Multiple Orders? The first question we investigate is whether it is beneficial to use multiple orders instead of a single order. For opportunistic uniform cost partitioning this change is beneficial in some domains and settings while it decreases coverage in others. This is due to opportunistic uniform cost partitioning distributing costs conservatively, which makes it less susceptible to the chosen order. Therefore, h^{OUCP} does not profit much from using multiple orders and using a single order is often preferable. For the other two order-dependent cost partitioning algorithms, greedy zero-one cost partitioning and saturated cost partitioning, using multiple diverse orders is clearly beneficial in all settings.

Question and Result	Comparison	HC	Sys	CART	COMB
multiple orders? → yes	$h_{\text{div}}^{\text{OUCP}}:h_{\text{one}}^{\text{OUCP}}$	0:0	1:4	7:3	1:2
	$h_{\text{div}}^{\text{GZOCP}}:h_{\text{one}}^{\text{GZOCP}}$	7:0	12:3	13:0	16:1
	$h_{\text{div}}^{\text{SCP}}:h_{\text{one}}^{\text{SCP}}$	8:0	12:1	17:0	17:2
reuse when not being greedy? → yes	$h_{\text{one}}^{\text{OUCP}}:h^{\text{UCP}}$	8:0	19:0	19:0	24:0
	$h_{\text{div}}^{\text{OUCP}}:h^{\text{UCP}}$	8:0	17:0	17:1	23:0
reuse when being greedy? → yes	$h_{\text{one}}^{\text{SCP}}:h_{\text{one}}^{\text{GZOCP}}$	7:1	20:0	23:0	27:1
	$h_{\text{div}}^{\text{SCP}}:h_{\text{div}}^{\text{GZOCP}}$	9:0	20:0	26:0	25:0
be greedy when not reusing? → yes	$h_{\text{one}}^{\text{GZOCP}}:h^{\text{UCP}}$	6:3	9:3	9:3	17:5
	$h_{\text{div}}^{\text{GZOCP}}:h^{\text{UCP}}$	10:1	16:3	17:1	25:0
be greedy when reusing? → yes	$h_{\text{one}}^{\text{SCP}}:h_{\text{one}}^{\text{OUCP}}$	4:4	17:3	19:1	26:2
	$h_{\text{div}}^{\text{SCP}}:h_{\text{div}}^{\text{OUCP}}$	9:1	21:1	26:1	30:0
strongest algorithm? → $h_{\text{div}}^{\text{SCP}}$	$h^{\text{CAN}}:h^{\text{PhO}}$	19:0	21:9	19:5	9:20
	$h_{\text{div}}^{\text{SCP}}:h^{\text{CAN}}$	6:1	23:1	31:0	35:0
	$h_{\text{div}}^{\text{SCP}}:h^{\text{PhO}}$	21:0	32:0	34:0	35:0

Table 11.4: Comparison of cost partitioning algorithms in four different settings: using hill climbing PDBs (HC), systematic PDBs (Sys), Cartesian abstraction heuristics (CART) and the combination of these three heuristic sets (COMB). For each comparison $X:Y$ a result $x:y$ states that in x out of 40 domains algorithm X solves more tasks than algorithm Y , while the opposite is true in y domains.

Reuse Costs? As predicted by the theoretical dominances (see Theorems 10.1 and 10.2), reusing costs is almost always beneficial in practice, regardless of the underlying heuristics, whether we use a single or multiple orders and whether we assign costs uniformly or greedily. There are a few domains where not reusing costs leads to a higher coverage, though. This might be unexpected, since reusing costs can never result in a weaker heuristic. However, this only holds if the cost partitioning algorithms use the same order(s). Since they use different greedy orders and optimization time limits, and therefore different orders, the resulting heuristics do not dominate each other.

Assign Costs Greedily? Assigning costs greedily (like h^{GZOC} and h^{SCP}) rather than uniformly (like h^{UCP} and h^{OUCP}) is also preferable in all settings. The trend is more pronounced when we do not reuse costs than when reusing them. Also, the difference becomes more apparent if we use multiple orders instead of a single order. If we focus on the comparison between $h_{\text{div}}^{\text{SCP}}$ and $h_{\text{div}}^{\text{OUCP}}$, we see that there are only three cases across all 40 domains and four heuristic settings where assigning costs uniformly is preferable¹. This suggests that opportunistic uniform cost partitioning distributes costs too conservatively and wastes lots of costs, even though there are component heuristics that could have profited from them. Assigning costs greedily, on the other hand, ensures that each component heuristic is allowed to use as much of the remaining costs as it needs.

Since saturated cost partitioning reuses costs and assigns them greedily it is the strongest cost partitioning algorithm evaluated so far, especially if we compute it for multiple diverse orders. We now evaluate the remaining two suboptimal cost partitioning algorithms, the canonical heuristic and post-hoc optimization, before comparing them to $h_{\text{div}}^{\text{SCP}}$.

Canonical Heuristic vs. Post-hoc Optimization Contrasting the theoretical dominance, the canonical heuristic solves more tasks than post-hoc optimization in many domains in all four settings. The opposite, however, is also true frequently.

To find out the reason why neither algorithm is preferable to the other in all settings, we inspect the reasons for why they fail to solve a task. Table 11.5 shows the failure reasons for a subset of cost partitioning algorithms on the set of tasks that are solved by at least one of the four algorithms. We can see that the canonical heuristic often fails to compute the heuristic value for the initial state in all four settings except for hill climbing PDBs. The reason for this depends on the setting: using systematic

¹ Tables A.1, A.2 and A.3 in the appendix reveal that for HC and Sys the domain in question is Hiking and for CART it is Mystery. In all three cases $h_{\text{div}}^{\text{OUCP}}$ solves only one task more than $h_{\text{div}}^{\text{SCP}}$

	computing $h(s_0, cost)$		during search	
	out of memory	out of time	out of memory	out of time
h_{div}^{SCP}	0/0/0/0	0/0/0/0	255/183/56/5	0/5/43/25
h^{CAN}	0/153/47/277	0/34/0/0	224/144/164/7	46/47/181/172
h^{PhO}	0/0/0/0	0/0/0/0	10/2/0/0	291/354/429/322
h^{OCP}	220/31/0/263	137/194/244/359	0/0/1/0	316/372/443/186

Table 11.5: Failure reasons for a subset of cost partitioning algorithms on the four different sets of abstraction heuristics. Each HC/Sys/CART/COMB entry states the number of tasks for which the given cost-partitioned heuristic ran out of memory or time before or during search using the respective set of heuristics. We only consider tasks that at least one of the four algorithms solves.

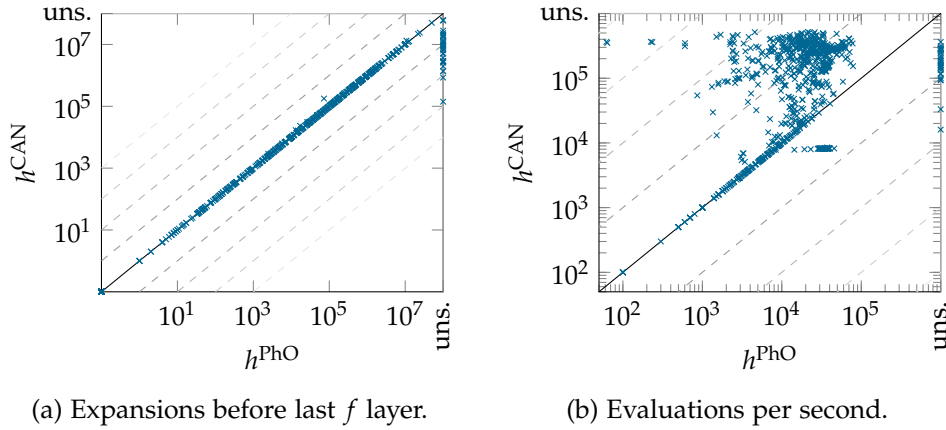
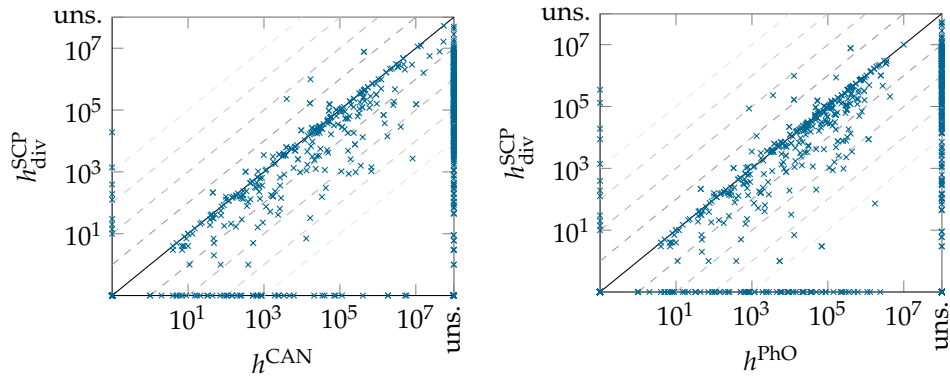


Figure 11.1: Canonical heuristic (h^{CAN}) vs. post-hoc optimization heuristic (h^{PhO}) using hill climbing PDBs. We draw tasks that are unsolved by one of the heuristics on the horizontal or vertical “uns.” line.

patterns, h^{CAN} runs out of memory for 153 tasks while computing maximal additive subsets and it runs out of time for 34 tasks while pruning dominated patterns. Using Cartesian abstractions and the combination of heuristics, dominance pruning is inapplicable and h^{CAN} exceeds the memory limit while computing maximal additive subsets for 47 and 277 tasks, respectively. In contrast, h^{PhO} is able to compute $h^{PhO}(s_0)$ for every task in all four settings. These results show that computing maximal additive subsets for a large number of component heuristics can become infeasible in practice due to the high memory requirements.

Using PDB heuristics for patterns found by hill climbing, both h^{CAN} and h^{PhO} are able to compute a heuristic value for the initial state in all tasks solved by at least one of the algorithms in Table 11.5. We therefore use this setting to compare the accuracy of the two heuristics. In Figure 11.1a



(a) Saturated cost partitioning ($h_{\text{div}}^{\text{SCP}}$) vs. canonical heuristic (h^{CAN}). (b) Saturated cost partitioning ($h_{\text{div}}^{\text{SCP}}$) vs. post-hoc optimization (h^{PhO}).

Figure 11.2: Number of expanded states before the last f layer using the combination of abstraction heuristics.

we can see that both heuristics need almost exactly the same amount of expansions for the commonly solved tasks. This finding for hill climbing PDBs is in line with the result by Pommerening et al. (2013), who showed that h^{CAN} is almost as accurate as h^{PhO} for *systematic* PDBs.

The reason why h^{CAN} solves more tasks than h^{PhO} using hill climbing PDBs lies in the difference in evaluation speed. Figure 11.1b compares the number of evaluations per second for the two heuristics. The canonical heuristic can be evaluated much faster than the post-hoc optimization heuristic, often by one and sometimes even by two orders of magnitude. We can also see evidence for this in Table 11.5: for unsolved tasks, h^{PhO} almost always runs out of time during search, while h^{CAN} also often runs out of memory.

Strongest Suboptimal Cost Partitioning Algorithm? We now compare h^{CAN} and h^{PhO} to $h_{\text{div}}^{\text{SCP}}$. Table 11.4 tells us that $h_{\text{div}}^{\text{SCP}}$ is almost always preferable to h^{CAN} and h^{PhO} in all settings. For example, using the combination of abstraction heuristics (COMB), $h_{\text{div}}^{\text{SCP}}$ has a higher coverage score than h^{CAN} and h^{PhO} in 35 out of 40 domains, while the opposite is never true.

To understand why $h_{\text{div}}^{\text{SCP}}$ has an edge over h^{CAN} and h^{PhO} in so many domains, we compare the number of expansions before the last f layer in the setting using the combination of abstraction heuristics. Figure 11.2a shows that for the majority of tasks $h_{\text{div}}^{\text{SCP}}$ is more accurate than h^{CAN} , often reducing the number of expanded states by several orders of magnitude. There are even 48 tasks (visible on the x axis) for which $h_{\text{div}}^{\text{SCP}}$, unlike h^{CAN} , makes no expansions before the last f layer. The converse is true for only 9 tasks.

	h^{UCP}	$h_{\text{one}}^{\text{OUCP}}$	$h_{\text{div}}^{\text{OUCP}}$	$h_{\text{one}}^{\text{GZOCP}}$	$h_{\text{div}}^{\text{GZOCP}}$	$h_{\text{one}}^{\text{SCP}}$	$h_{\text{div}}^{\text{SCP}}$	h^{CAN}	h^{PhO}	h^{OCP}
HC	807	819	819	813	822	818	838	823	792	420
SYS	743	791	789	769	789	867	905	715	737	496
CART	696	756	760	739	758	881	994	701	664	405
COMB	739	812	811	799	826	977	1063	637	771	285

Table 11.6: Total number of solved tasks by different cost partitioning algorithms using different sets of heuristics: hill climbing PDBs (HC), systematic PDBs (Sys), Cartesian abstraction heuristics (CART) and the combination of these three heuristic sets (COMB). For each heuristic setting we highlight the algorithm with the maximum coverage score in bold.

We see a similar picture when we compare the number of expanded states by $h_{\text{div}}^{\text{SCP}}$ and h^{PhO} in Figure 11.2b. $h_{\text{div}}^{\text{SCP}}$ is more accurate than h^{PhO} for the majority of tasks. Again, $h_{\text{div}}^{\text{SCP}}$ is perfect more often (75 tasks) than h^{PhO} (12 tasks) when the other heuristic is not perfect. In addition to being more accurate than h^{CAN} and h^{PhO} , $h_{\text{div}}^{\text{SCP}}$ is also faster to evaluate than both other heuristics. The row for $h_{\text{div}}^{\text{SCP}}$ in Table 11.5 hints at this result: out of the four compared algorithms in the table, $h_{\text{div}}^{\text{SCP}}$ runs out of time least often.

Total Coverage In Table 11.6, we compare the total coverage scores for all cost partitioning algorithms in the four abstraction heuristic settings. Saturated cost partitioning using diverse orders not only has an edge over all other suboptimal cost partitioning algorithms in a domain-wise comparison (see Table 11.4 and Tables A.1, A.2, A.3 and A.4 in the appendix), but Table 11.6 reveals that it also achieves the highest total coverage scores in all four settings. Except for the setting of hill climbing PDBs, even saturated cost partitioning using only a single order ($h_{\text{one}}^{\text{SCP}}$) solves more tasks in total than all other cost partitioning algorithms.

Optimal Cost Partitioning The last question we want to answer in this section is how optimal cost partitioning compares against the suboptimal cost partitioning algorithms. We begin by analyzing how closely the most accurate suboptimal cost partitioning algorithm, $h_{\text{div}}^{\text{SCP}}$, approximates the heuristic quality of optimal cost partitioning. For this comparison we use the combination of abstraction heuristics. Since $h_{\text{div}}^{\text{SCP}}$ is much faster to evaluate than h^{OCP} , $h_{\text{div}}^{\text{SCP}}$ has a significantly higher coverage (1063 tasks) than h^{OCP} (272 tasks). Even if we raise the time limit for optimal cost partitioning to 24 hours ($h_{24\text{h}}^{\text{OCP}}$), it only solves 395 tasks.

Figure 11.3 compares the number of expansions needed by $h_{\text{div}}^{\text{SCP}}$ and $h_{24\text{h}}^{\text{OCP}}$. The 1667 benchmark tasks can be divided into the following groups:

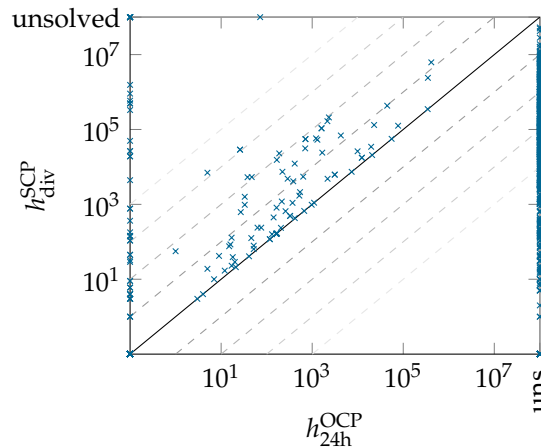


Figure 11.3: Number of expansions before the last f layer for h_{24h}^{OCP} and h_{div}^{SCP} , both computed over the combination of abstraction heuristics.

583 tasks are solved by neither heuristic. For 7 tasks both heuristics detect unsolvability and for 2 tasks only h_{24h}^{OCP} is able to do so. There are 680 tasks solved by h_{div}^{SCP} which h_{24h}^{OCP} fails to solve and for 12 tasks the opposite is the case. For 257 commonly solved tasks, both heuristics are perfect, i.e., they need no expansions before the last f layer. There are 41 commonly solved tasks for which h_{24h}^{OCP} is perfect, but h_{div}^{SCP} is not and 85 commonly solved tasks for which neither heuristic is perfect.

h_{div}^{SCP} needs more expansions than h_{24h}^{OCP} in 113 of the 383 commonly solved tasks, revealing that h_{div}^{SCP} is often not as accurate as h_{24h}^{OCP} . This shows that one can still hope to find better approximations of optimal cost partitionings. However, the question is whether better approximations can be computed fast enough or whether the additional computation time outweighs the increase in accuracy.

In Table 11.6 we can see that optimal cost partitioning has the lowest total coverage in all settings, failing to come even close to the coverage scores of the other cost partitioning algorithms. We find the reason for this in Table 11.5. Optimal cost partitioning very often runs out of memory while computing the heuristic value for the initial state of tasks that are solved by other cost partitioning algorithms. However, due to their small size, this never happens when using Cartesian abstractions. Time is also frequently a limiting factor in all four settings for computing $h^{OCP}(s_0, cost)$. If the computation finishes and the search starts, h^{OCP} almost never hits the memory limit, showing that evaluating the heuristic takes very long. Taken together, evaluating the optimal cost partitioning heuristic is too slow and memory-consuming for h^{OCP} to be a viable alternative for explicitly represented abstraction heuristics in practice.

	h^{UCP}	$h_{\text{one}}^{\text{OUCP}}$	$h_{\text{one}}^{\text{GZOCP}}$	$h_{\text{one}}^{\text{SCP}}$	h^{CAN}	h^{PhO}	h^{OCP}	Coverage
h^{UCP}	–	4	7	2	12	24	26	899
$h_{\text{one}}^{\text{OUCP}}$	1	–	5	0	12	25	26	892
$h_{\text{one}}^{\text{GZOCP}}$	5	5	–	0	11	25	26	888
$h_{\text{one}}^{\text{SCP}}$	9	10	10	–	15	30	30	932
h^{CAN}	4	4	6	1	–	23	25	857
h^{PhO}	2	2	4	0	4	–	8	830
h^{OCP}	1	1	2	0	4	1	–	820

Table 11.7: Left: Pairwise comparison of cost partitioning algorithms using landmark heuristics. The entry in row r and column c holds the number of domains in which algorithm r solved more tasks than algorithm c . For each algorithm pair we highlight the maximum of the entries (r, c) and (c, r) in bold. Right: Total number of solved tasks by each algorithm.

11.3 Landmark Heuristics

We now compare cost partitioning algorithms for landmark heuristics. To the best of our knowledge only two ways of combining landmark heuristics admissibly have been previously evaluated: optimal and uniform cost partitioning (Karpas and Domshlak 2009). In contrast to the experiments above, for landmark heuristics we cannot precompute cost partitionings before the search starts, since the set of unachieved landmarks changes between evaluated states. We therefore use a single greedy order for all order-dependent cost partitioning algorithms: as for abstraction heuristics, $h_{\text{one}}^{\text{OUCP}}$ uses the *stolen* greedy variant, $h_{\text{one}}^{\text{GZOCP}}$ uses h and $h_{\text{one}}^{\text{SCP}}$ uses $\frac{h}{\text{stolen}}$.

Table 11.7 compares the different cost partitioning algorithms for the landmark heuristics computed by the BJOLP planner (Domshlak et al. 2011). (For coverage scores in individual domains see Table A.5 in the appendix.) As in the four settings using abstraction heuristics, all suboptimal cost partitioning algorithms outperform optimal cost partitioning. In difference to the results above, however, h^{OCP} comes closer to the other cost partitioning algorithms in terms of total coverage. This is the case since the linear programs that have to be solved for optimally partitioning costs among landmark heuristics are usually much smaller than the ones for abstraction heuristics.

Comparing the suboptimal cost partitioning algorithms, we notice that h^{UCP} solves more tasks than $h_{\text{one}}^{\text{OUCP}}$ in four domains, even though $h_{\text{one}}^{\text{OUCP}}$ dominates h^{UCP} theoretically. This is the case since computing landmark

orders and keeping track of the remaining cost function makes h^{OUCP} substantially slower to evaluate than h^{UCP} . On the other hand, only a few tasks actually benefit from reusing costs, since the component landmark heuristics usually consume all of the costs they are offered. As a result, reusing costs is not worth the extra computational effort when distributing costs uniformly among landmark heuristics.

For greedy cost assignments, however, reusing costs is very helpful: $h_{\text{one}}^{\text{SCP}}$ solves more tasks than $h_{\text{one}}^{\text{GZOCP}}$ in 10 domains, while the opposite is never true. Uniform cost partitioning has a slight edge over greedy cost assignment if costs are not reused (h^{UCP} vs. $h_{\text{one}}^{\text{GZOCP}}$: 7 to 5). However, distributing costs greedily is clearly beneficial when reusing costs ($h_{\text{one}}^{\text{SCP}}$ vs. $h_{\text{one}}^{\text{OUCP}}$: 10 to 0). If we distribute costs greedily *and* reuse unused costs, as done by $h_{\text{one}}^{\text{SCP}}$, the extra computational effort compared to h^{UCP} clearly leads to a more accurate heuristic: $h_{\text{one}}^{\text{SCP}}$ solves more tasks than h^{UCP} in 9 domains, while the opposite is the case in only 2 domains. Neither h^{CAN} nor h^{PhO} are able to come close to the other suboptimal cost partitioning algorithms in terms of domain-wise or total coverage.

Overall, the differences between the results for different cost partitioning algorithms are smaller for landmark heuristics than for the abstraction heuristics above. This stems from the fact that there are fewer choices to make when partitioning costs for landmark heuristics. However, saturated cost partitioning again has an edge over the other algorithms in a per-domain comparison and solves the highest number of tasks in total (932 tasks).

11.4 Comparison of Different Approaches

In the experiments above, we compared different cost partitioning algorithms operating on the same set of heuristics. In all settings $h_{\text{div}}^{\text{SCP}}$ (respectively $h_{\text{one}}^{\text{SCP}}$ for landmark heuristics) had an edge over all other cost partitioning algorithms. Comparing these five planners based on saturated cost partitioning (named $h_{\text{HC}}^{\text{SCP}}$, $h_{\text{SYS}}^{\text{SCP}}$, $h_{\text{CART}}^{\text{SCP}}$, $h_{\text{COMB}}^{\text{SCP}}$ and $h_{\text{LM}}^{\text{SCP}}$ in the following) allows us to shed some light on the relative accuracy of the underlying heuristics. In Table 11.8, we compare the h^{SCP} -based planners to each other and to five of the strongest admissible heuristics from the literature (see Table A.6 in the appendix for domain-wise coverage scores):

- $h^{\text{LM-cut}}$: the landmark-cut heuristic (Helmert and Domshlak 2009)
- h^{iPDB} : the canonical heuristic using pattern databases found by 15 minutes of hill climbing (Haslum et al. 2007; Sievers et al. 2012)

	$h_{\text{HC}}^{\text{SCP}}$	$h_{\text{SYS}}^{\text{SCP}}$	$h_{\text{CART}}^{\text{SCP}}$	$h_{\text{COMB}}^{\text{SCP}}$	$h_{\text{LM}}^{\text{SCP}}$	$h^{\text{LM-cut}}$	h^{iPDB}	$h^{\text{M\&S}}$	$h_{\text{SYS}}^{\text{PhO}}$	h^{SEQ}	Coverage
$h_{\text{HC}}^{\text{SCP}}$	-	3	8	0	15	18	3	18	28	23	838
$h_{\text{SYS}}^{\text{SCP}}$	15	-	14	2	22	24	10	22	32	26	905
$h_{\text{CART}}^{\text{SCP}}$	17	14	-	3	20	23	15	23	33	29	994
$h_{\text{COMB}}^{\text{SCP}}$	26	20	19	-	28	28	21	27	37	30	1063
$h_{\text{LM}}^{\text{SCP}}$	10	8	3	1	-	15	8	16	22	23	932
$h^{\text{LM-cut}}$	12	9	4	2	12	-	10	14	25	24	881
h^{iPDB}	10	7	9	2	19	22	-	19	31	24	881
$h^{\text{M\&S}}$	10	8	6	2	15	15	8	-	24	21	808
$h_{\text{SYS}}^{\text{PhO}}$	4	0	2	0	4	3	1	7	-	16	737
h^{SEQ}	8	5	6	2	7	6	7	8	13	-	735

Table 11.8: Left: Domain-wise coverage comparison of different heuristics. The entry in row r and column c holds the number of domains in which heuristic r solved more tasks than heuristic c . For each heuristic pair we highlight the maximum of the entries (r, c) and (c, r) in bold. Right: Total number of solved tasks by each heuristic.

- $h^{\text{M\&S}}$: merge-and-shrink using bisimulation, the SCC-DFP merge strategy and at most 50 000 abstract states (Helmert et al. 2014; Sievers et al. 2016)
- $h_{\text{SYS}}^{\text{PhO}}$: post-hoc optimization using systematic patterns of sizes 1 and 2 (Pommerening et al. 2013)
- h^{SEQ} : the state-equation heuristic (Bonet 2013)

Saturated Cost Partitioning Heuristics Inspecting the results for the five h^{SCP} -based planners, we see that the landmarks-based $h_{\text{LM}}^{\text{SCP}}$ usually solves fewer tasks per domain than the four planners based on abstraction heuristics. $h_{\text{SYS}}^{\text{SCP}}$ and $h_{\text{CART}}^{\text{SCP}}$ outperform the respective other heuristic in 14 domains each. Both of them have an edge over $h_{\text{HC}}^{\text{SCP}}$ in domain-wise and overall comparisons. As expected, computing cost partitionings over the combined set of abstraction heuristics leads to a much stronger heuristic ($h_{\text{COMB}}^{\text{SCP}}$) than using the subsets of heuristics. $h_{\text{COMB}}^{\text{SCP}}$ solves as many or more tasks than the other h^{SCP} -based planners in almost all domains and has the highest total coverage score (1063 tasks).

Other Admissible Heuristics In a domain-wise comparison, $h^{\text{LM-cut}}$, $h^{\text{M\&S}}$, $h_{\text{SYS}}^{\text{PhO}}$ and h^{SEQ} are outperformed by all saturated cost partitioning

heuristics. Only h^{iPDB} has an edge over $h_{\text{HC}}^{\text{SCP}}$ and $h_{\text{LM}}^{\text{SCP}}$ in more domains than the other way around. $h^{\text{LM-cut}}$ and h^{iPDB} solve more tasks in total (881 tasks) than $h_{\text{HC}}^{\text{SCP}}$ (838 tasks), but in all other comparisons between h^{SCP} -based and other heuristics, the saturated cost partitioning heuristics have a higher total coverage.

Cartesian Abstractions Table 11.8 also holds further evidence for the finding that using good orders is critical for saturated cost partitioning heuristics. The $h_{\text{LM+s}_*}^{\text{CEGAR}}$ heuristic in Table 7.1 and the $h_{\text{CART}}^{\text{SCP}}$ heuristic in Table 11.8 both use Cartesian abstractions of landmark and goal task decompositions. While $h_{\text{LM+s}_*}^{\text{CEGAR}}$ orders the resulting component heuristics randomly and solves 798 tasks, $h_{\text{CART}}^{\text{SCP}}$ uses a diverse set of optimized greedy orders and solves 994 tasks, 196 more tasks than $h_{\text{LM+s}_*}^{\text{CEGAR}}$. We saw in Table 7.1 that $h_{\text{LM+s}_*}^{\text{CEGAR}}$ has a lower total coverage score than h^{iPDB} and $h^{\text{M\&S}}$. However, $h_{\text{CART}}^{\text{SCP}}$ (solving 994 tasks) not only outperforms the three non-CEGAR heuristics in Table 7.1, h^{iPDB} (881 tasks), $h^{\text{M\&S}}$ (808 tasks) and $h_{\text{SYS}}^{\text{PhO}}$ (737 tasks), but also $h^{\text{LM-cut}}$ (881 tasks) and h^{SEQ} (735 tasks). The domain-wise comparison shows that $h_{\text{CART}}^{\text{SCP}}$ solves more tasks than $h^{\text{LM-cut}}$, h^{iPDB} , $h^{\text{M\&S}}$, $h_{\text{SYS}}^{\text{PhO}}$ and h^{SEQ} in 23, 15, 23, 33 and 29 domains, while the opposite is true in only 4, 9, 6, 2 and 6 domains, respectively. These results show that we are able to outperform the formerly strongest admissible heuristics for optimal classical planning by only using methods introduced in this thesis: counterexample-guided Cartesian abstraction refinement and saturated cost partitioning.

Combined Abstractions Combining all abstraction heuristics in $h_{\text{COMB}}^{\text{SCP}}$ results in an even stronger heuristic than $h_{\text{CART}}^{\text{SCP}}$. It outperforms all heuristics that are not based on saturated cost partitioning in Table 11.8, solving 182 more tasks than the two non- h^{SCP} -based heuristics with the highest total coverage, $h^{\text{LM-cut}}$ and h^{iPDB} . Beyond total coverage, $h_{\text{COMB}}^{\text{SCP}}$ also has an edge over the other heuristics in most individual domains. Out of the 40 domains, $h_{\text{COMB}}^{\text{SCP}}$ solves more tasks than $h^{\text{LM-cut}}$, h^{iPDB} , $h^{\text{M\&S}}$, $h_{\text{SYS}}^{\text{PhO}}$ and h^{SEQ} in 28, 21, 27, 37 and 30 domains, respectively, while the opposite is true in at most 2 domains for all five heuristics. In summary, $h_{\text{COMB}}^{\text{SCP}}$ is stronger than any other heuristic based on saturated cost partitioning, cost partitioning in general and all other evaluated heuristics.

SymBA* This raises the question how $h_{\text{COMB}}^{\text{SCP}}$ compares to the winner of the IPC 2014 sequential optimization track, the symbolic search planner SymBA₂* (Torralba et al. 2016). To allow for an unbiased comparison, we evaluate a version of $h_{\text{COMB}}^{\text{SCP}}$ that uses h^2 mutexes to prune irrelevant operators (Alcázar and Torralba 2015), a preprocessing technique that is an important ingredient of SymBA₂* and can be combined with any planning

algorithm. Table A.7 in the appendix holds coverage results. In 23 domains $h_{\text{COMB}}^{\text{SCP}}$ using h^2 mutexes solves more tasks than SymBA_2^* , while the opposite is true in only 10 domains. $h_{\text{COMB}}^{\text{SCP}}$ using h^2 mutexes also has an edge over SymBA_2^* in terms of total coverage: while SymBA_2^* solves 1027 tasks, $h_{\text{COMB}}^{\text{SCP}}$ finds a solution for 1115 tasks, a difference of 88 tasks. Since task difficulty tends to scale exponentially in optimal classical planning, this increase in coverage compared to the previous state of the art is remarkable.

Part IV

CONCLUSION

CONCLUSION

In Part I, we introduced a CEGAR approach for classical planning and showed that it delivers promising performance. We believe that further performance improvements are possible through speed optimizations in the refinement loop, which will enable larger abstractions to be generated in reasonable time. One possibility is to refute not one but all optimal solutions in one iteration. This should shift a big proportion of the time needed to build the abstraction from looking for abstract solutions to actually refining the abstraction.

We often have many options for selecting the variable on which to split and how to partition its values. Therefore, another approach for improving the resulting Cartesian abstractions could be to investigate the choice of refinement strategy in more depth. For example, it could be beneficial to always choose the split that increases the estimated accuracy of the resulting heuristic the most.

As is the case for pattern databases, switching from one Cartesian abstraction to multiple Cartesian abstractions is highly beneficial. We showed that constructing diverse sets of abstractions and combining them with saturated cost partitioning yields heuristics that outperform single Cartesian abstractions and are competitive with abstraction heuristics from the literature.

We believe that Cartesian abstractions, counterexample-guided abstraction refinement and the task decomposition methods are useful concepts that can contribute to the further development of strong abstraction heuristics for automated planning.

In Part II, we showed both theoretically and in experiments that the order in which saturated cost partitioning considers a set of component heuristics greatly influences the quality of the resulting cost-partitioned heuristic. Greedy orders result in significantly more accurate heuristics than those obtained with random orders. In addition, greedy orders greatly benefit from optimization via a hill-climbing search. Maximizing over heuristics from multiple orders leads to further improvements, especially when explicitly diversifying the set of orders to include only those that prove useful on a set of sample states.

As our experiments demonstrated, hill climbing in the space of orders often keeps finding better orders even after one thousand seconds. Therefore, it could be beneficial to continue searching for more efficient order generation methods.

In Part III, we presented the first systematic theoretical and experimental comparison of cost partitioning algorithms for optimal classical planning. Among several other dominance and non-dominance results, our theoretical analysis showed that saturated cost partitioning dominates zero-one cost partitioning and suggested a new cost partitioning algorithm called opportunistic uniform cost partitioning, which dominates uniform cost partitioning.

The fact that the concept of cost saturation is applicable to greedy zero-one cost partitioning (yielding saturated cost partitioning) and uniform cost partitioning (yielding opportunistic uniform cost partitioning), raises the question of whether there are other cost partitioning algorithms that can profit from cost saturation. We believe that this could be the case for post-hoc optimization. More generally, future research could investigate whether there are further connections or synergies between different cost partitioning algorithms.

Our experimental evaluation revealed that it is almost always beneficial to use multiple orders for order-dependent cost partitioning algorithms, to assign costs greedily and to reuse unconsumed costs. We also showed that saturated cost partitioning is the method of choice in all tested settings, outperforming the previous best cost partitioning methods for all evaluated heuristics.

However, we also saw that there is still a gap in accuracy between the strongest suboptimal cost partitioning algorithm, saturated cost partitioning, and optimal cost partitioning. The question is whether and how we can narrow this gap. Most cost partitioning algorithms use a one-shot method for generating each cost function. Only the linear program solvers underlying optimal cost partitioning for abstraction heuristics and post-hoc optimization *iteratively* obtain better cost partitionings until finding the one maximizing the objective function. Optimizing an order via hill climbing changes the cost partitioning over time by repeatedly computing a new cost partitioning from scratch. We believe that future research could try to come up with other methods that iteratively improve an existing cost partitioning, possibly converging to the optimal cost partitioning but using significantly less time and memory.

Finally, having shown that saturated cost partitioning works well for abstraction and landmark heuristics, future research could try to find (non-trivial) saturators for other types of heuristics, allowing to use these heuristics for saturated cost partitioning.

APPENDIX

A.1 Pseudo-code for CEGAR Informed Transition Check

Algorithm 10 Rewiring of incoming transitions. The refinement in progress splits state $[s]$ into states d and e on variable v . For the old transition $a \xrightarrow{o} [s]$ this procedure adds new transitions from a to the new states d and e where necessary.

```

1: procedure REWIREINCOMINGTRANSITION( $a \xrightarrow{o} [s], d, e, v$ )
2:   if  $v \notin \text{vars}(\text{post}(o))$  then
3:     if  $|\text{dom}(v, a) \cap \text{dom}(v, d)| \neq \emptyset$  then
4:       ADDTRANSITION( $a, o, d$ )
5:     if  $|\text{dom}(v, a) \cap \text{dom}(v, e)| \neq \emptyset$  then
6:       ADDTRANSITION( $a, o, e$ )
7:   else if  $\text{post}(o)[v] \in \text{dom}(v, d)$  then
8:     ADDTRANSITION( $a, o, d$ )
9:   else
10:    ADDTRANSITION( $a, o, e$ )

```

Algorithm 11 Rewiring of outgoing transitions. The refinement in progress splits state $[s]$ into states d and e on variable v . For the old transition $[s] \xrightarrow{o} b$ this procedure adds new transitions from the new states d and e to b where necessary.

```

1: procedure REWIREOUTGOINGTRANSITION( $[s] \xrightarrow{o} b, d, e, v$ )
2:   if  $v \notin \text{vars}(\text{post}(o))$  then
3:     if  $|\text{dom}(v, d) \cap \text{dom}(v, b)| \neq \emptyset$  then
4:       ADDTRANSITION( $d, o, b$ )
5:     if  $|\text{dom}(v, e) \cap \text{dom}(v, b)| \neq \emptyset$  then
6:       ADDTRANSITION( $e, o, b$ )
7:   else if  $v \notin \text{vars}(\text{pre}(o))$  then
8:     ADDTRANSITION( $d, o, b$ )
9:     ADDTRANSITION( $e, o, b$ )
10:  else if  $\text{pre}(o)[v] \in \text{dom}(v, e)$  then
11:    ADDTRANSITION( $e, o, b$ )
12:  else
13:    ADDTRANSITION( $d, o, b$ )

```

Algorithm 12 Rewiring of self-loops. The refinement in progress splits state $[s]$ into states d and e on variable v . This procedure adds new self-loops and/or transitions between the new states d and e for the old self-loop $[s] \xrightarrow{o} [s]$ where necessary.

```

1: procedure REWIRESELFLOOP( $[s] \xrightarrow{o} [s], d, e, v$ )
2:   if  $v \notin \text{vars}(\text{pre}(o))$  then
3:     if  $v \notin \text{vars}(\text{post}(o))$  then
4:       ADDSELFLOOP( $d, o$ )
5:       ADDSELFLOOP( $e, o$ )
6:     else if  $\text{post}(o)[v] \in \text{dom}(v, e)$  then
7:       ADDTRANSITION( $d, o, e$ )
8:       ADDSELFLOOP( $e, o$ )
9:     else
10:      ADDSELFLOOP( $d, o$ )
11:      ADDTRANSITION( $e, o, d$ )
12:   else if  $\text{pre}(o)[v] \in \text{dom}(v, e)$  then
13:     if  $\text{post}(o)[v] \in \text{dom}(v, e)$  then
14:       ADDSELFLOOP( $e, o$ )
15:     else
16:       ADDTRANSITION( $e, o, d$ )
17:   else
18:     if  $\text{post}(o)[v] \in \text{dom}(v, e)$  then
19:       ADDTRANSITION( $d, o, e$ )
20:     else
21:       ADDSELFLOOP( $d, o$ )

```

A.2 Detailed Results for Cost Partitioning Algorithms

	h^{UCP}	$h_{\text{one}}^{\text{OUCP}}$	$h_{\text{div}}^{\text{OUCP}}$	$h_{\text{one}}^{\text{GZOCP}}$	$h_{\text{div}}^{\text{GZOCP}}$	$h_{\text{one}}^{\text{SCP}}$	$h_{\text{div}}^{\text{SCP}}$	h^{CAN}	h^{PhO}	h^{OCP}
airport ₍₅₀₎	24	24	24	24	24	24	24	24	23	23
barman ₍₃₄₎	4	4	4	4	4	4	4	4	4	0
blocks ₍₃₅₎	28	28	28	28	28	28	28	28	28	26
childsnack ₍₂₀₎	0	0	0	0	0	0	0	0	0	0
depot ₍₂₂₎	7	7	7	7	7	7	7	7	7	4
driverlog ₍₂₀₎	13	13	13	14	14	14	14	14	13	2
elevators ₍₅₀₎	40	40	40	38	40	38	41	41	40	1
floortile ₍₄₀₎	2	3	3	3	3	5	5	2	2	0
freecell ₍₈₀₎	20	21	21	20	20	21	21	20	19	0
ged ₍₂₀₎	19	19	19	19	19	15	19	19	19	0
grid ₍₅₎	3	3	3	3	3	3	3	3	3	1
gripper ₍₂₀₎	8	8	8	8	8	8	8	8	8	6
hiking ₍₂₀₎	13	13	13	12	12	12	12	12	11	2
logistics ₍₆₃₎	27	27	27	27	27	27	27	28	27	4
miconic ₍₁₅₀₎	63	65	65	65	65	65	65	65	63	20
movie ₍₃₀₎	30	30	30	30	30	30	30	30	30	30
mprime ₍₃₅₎	24	24	24	23	24	24	24	24	22	16
mystery ₍₃₀₎	17	17	17	17	17	17	17	17	15	9
nomystery ₍₂₀₎	19	19	19	19	20	19	20	20	19	2
openstacks ₍₁₀₀₎	49	49	49	49	49	49	49	49	49	27
parcprinter ₍₅₀₎	28	28	28	28	28	28	28	28	28	26
parking ₍₄₀₎	13	13	13	13	13	13	13	13	10	3
pathways ₍₃₀₎	4	4	4	4	4	4	4	4	4	4
pegsol ₍₅₀₎	44	44	44	44	44	44	44	44	44	27
pipes-nt ₍₅₀₎	20	20	20	20	20	20	20	20	19	14
pipes-t ₍₅₀₎	14	16	16	14	14	15	16	13	11	2
psr-small ₍₅₀₎	49	50	50	49	50	50	50	50	49	41
rovers ₍₄₀₎	7	8	8	8	8	8	8	8	7	5
satellite ₍₃₆₎	6	6	6	6	6	6	6	6	6	4
scanalyzer ₍₅₀₎	21	21	21	23	23	23	27	23	23	6
sokoban ₍₅₀₎	46	46	46	46	46	46	46	46	46	39
storage ₍₃₀₎	16	16	16	16	16	16	16	16	15	10
tetris ₍₁₇₎	9	9	9	9	9	9	10	9	5	6
tidybot ₍₄₀₎	22	22	22	22	22	22	22	22	22	8
tpp ₍₃₀₎	6	6	6	6	6	6	6	6	6	6
transport ₍₇₀₎	30	33	33	33	34	33	35	35	31	6
trucks ₍₃₀₎	9	9	9	9	9	9	9	9	8	4
visitall ₍₄₀₎	26	26	26	26	26	28	28	28	28	19
woodwork ₍₅₀₎	15	15	15	15	17	15	19	15	15	15
zenotravel ₍₂₀₎	12	13	13	12	13	13	13	13	13	2
Sum ₍₁₆₆₇₎	807	819	819	813	822	818	838	823	792	420

Table A.1: Number of solved tasks using PDB heuristics for hill climbing patterns.

	h^{UCP}	h^{OUCP}_{one}	h^{OUCP}_{div}	h^{GZOCP}_{one}	h^{GZOCP}_{div}	h^{SCP}_{one}	h^{SCP}_{div}	h^{CAN}	h^{PhO}	h^{OCP}
airport ⁽⁵⁰⁾	23	24	24	26	24	26	28	20	27	16
barman ⁽³⁴⁾	4	4	4	4	4	4	4	4	4	0
blocks ⁽³⁵⁾	21	26	25	26	26	28	28	28	26	15
childsnack ⁽²⁰⁾	0	0	0	0	0	0	0	0	0	0
depot ⁽²²⁾	7	7	7	8	9	12	12	7	7	2
driverlog ⁽²⁰⁾	13	13	13	13	13	14	15	13	13	10
elevators ⁽⁵⁰⁾	31	31	31	31	36	31	36	38	36	14
floortile ⁽⁴⁰⁾	2	3	3	2	2	4	5	2	2	2
freecell ⁽⁸⁰⁾	20	21	20	20	21	21	21	20	15	1
ged ⁽²⁰⁾	19	19	19	15	18	15	19	19	15	3
grid ⁽⁵⁾	2	3	3	2	3	2	3	2	2	0
gripper ⁽²⁰⁾	8	8	8	8	8	8	8	8	7	4
hiking ⁽²⁰⁾	12	13	13	12	12	12	12	12	11	4
logistics ⁽⁶³⁾	25	25	25	25	27	27	33	22	26	34
miconic ⁽¹⁵⁰⁾	55	58	60	55	55	65	68	55	54	35
movie ⁽³⁰⁾	30	30	30	30	30	30	30	30	30	30
mprime ⁽³⁵⁾	23	26	26	23	23	29	30	23	21	18
mystery ⁽³⁰⁾	17	18	18	17	16	18	18	16	15	11
nomystery ⁽²⁰⁾	16	16	16	16	20	16	20	20	16	8
openstacks ⁽¹⁰⁰⁾	49	49	49	49	49	49	49	30	47	12
parcprinter ⁽⁵⁰⁾	26	33	32	36	38	39	38	14	30	49
parking ⁽⁴⁰⁾	2	13	13	1	1	13	13	0	3	0
pathways ⁽³⁰⁾	4	4	4	4	4	4	4	4	4	4
pegsol ⁽⁵⁰⁾	44	44	44	44	44	48	48	32	44	15
pipes-nt ⁽⁵⁰⁾	19	20	20	19	20	20	20	17	15	9
pipes-t ⁽⁵⁰⁾	14	16	16	15	15	17	17	15	9	2
psr-small ⁽⁵⁰⁾	49	50	50	49	49	50	50	48	49	49
rovers ⁽⁴⁰⁾	7	7	7	7	7	7	7	7	7	6
satellite ⁽³⁶⁾	6	6	6	6	6	6	6	4	6	5
scanalyzer ⁽⁵⁰⁾	21	21	21	21	21	27	33	23	11	6
sokoban ⁽⁵⁰⁾	48	50	50	50	48	50	50	50	49	14
storage ⁽³⁰⁾	15	16	15	16	16	16	16	16	15	9
tetris ⁽¹⁷⁾	9	9	9	9	9	11	11	4	3	0
tidybot ⁽⁴⁰⁾	22	22	22	22	22	22	22	22	21	8
tpp ⁽³⁰⁾	6	6	6	6	6	6	6	6	6	6
transport ⁽⁷⁰⁾	24	24	24	24	24	24	24	24	21	18
trucks ⁽³⁰⁾	8	8	8	9	9	9	9	9	7	5
visitall ⁽⁴⁰⁾	12	13	13	12	13	30	30	28	27	25
woodwork ⁽⁵⁰⁾	19	23	23	27	29	44	49	11	25	39
zenotravel ⁽²⁰⁾	11	12	12	10	12	13	13	12	11	8
Sum ⁽¹⁶⁶⁷⁾	743	791	789	769	789	867	905	715	737	496

Table A.2: Number of solved tasks using PDB heuristics for systematic patterns.

	h^{UCP}	h^{OUCP}_{one}	h^{OUCP}_{div}	h^{GZOCP}_{one}	h^{GZOCP}_{div}	h^{SCP}_{one}	h^{SCP}_{div}	h^{CAN}	h^{PhO}	h^{OCP}
airport ₍₅₀₎	22	29	29	31	31	31	31	25	27	16
barman ₍₃₄₎	4	4	4	4	4	4	4	4	4	0
blocks ₍₃₅₎	18	18	18	18	18	25	28	18	18	9
childsnaek ₍₂₀₎	0	0	0	0	0	0	0	0	0	0
depot ₍₂₂₎	5	6	7	5	7	11	11	7	5	1
driverlog ₍₂₀₎	12	13	13	10	13	14	14	9	8	9
elevators ₍₅₀₎	33	37	39	35	37	39	44	37	37	8
floortile ₍₄₀₎	2	2	2	2	2	2	3	2	2	1
freecell ₍₈₀₎	20	24	20	20	20	57	65	25	16	6
ged ₍₂₀₎	15	15	15	15	15	15	15	15	15	6
grid ₍₅₎	2	3	3	2	2	3	3	2	2	1
gripper ₍₂₀₎	8	8	7	8	8	8	8	8	7	5
hiking ₍₂₀₎	12	13	12	12	12	13	14	12	11	3
logistics ₍₆₃₎	17	27	28	25	25	29	39	16	17	28
miconic ₍₁₅₀₎	55	60	61	55	56	84	142	50	52	52
movie ₍₃₀₎	30	30	30	30	30	30	30	30	30	30
mprime ₍₃₅₎	27	27	27	27	27	27	27	27	26	16
mystery ₍₃₀₎	18	18	18	17	17	17	17	17	17	11
nomystery ₍₂₀₎	10	18	20	16	17	20	20	12	10	6
openstacks ₍₁₀₀₎	49	49	49	49	49	51	51	49	47	7
parcprinter ₍₅₀₎	18	24	26	32	34	35	38	18	24	12
parking ₍₄₀₎	0	0	0	0	0	0	10	0	0	0
pathways ₍₃₀₎	4	4	4	4	4	4	5	4	4	4
pegsol ₍₅₀₎	44	44	44	44	46	48	48	46	44	14
pipes-nt ₍₅₀₎	17	18	18	17	18	22	23	18	16	7
pipes-t ₍₅₀₎	13	14	14	13	14	15	16	14	10	2
psr-small ₍₅₀₎	49	49	49	49	49	49	49	49	49	43
rovers ₍₄₀₎	7	8	8	7	8	8	8	6	6	6
satellite ₍₃₆₎	6	6	6	6	6	7	7	6	6	3
scanalyzer ₍₅₀₎	21	21	21	21	21	23	23	21	15	5
sokoban ₍₅₀₎	39	39	39	41	41	46	46	37	39	11
storage ₍₃₀₎	16	16	16	16	16	16	16	15	14	13
tetris ₍₁₇₎	9	9	9	9	9	9	9	9	7	2
tidybot ₍₄₀₎	22	22	22	22	22	22	22	21	16	2
tpp ₍₃₀₎	6	7	7	7	8	7	8	6	6	6
transport ₍₇₀₎	24	24	24	24	24	24	25	24	22	3
trucks ₍₃₀₎	6	10	10	9	9	12	12	8	6	4
visitall ₍₄₀₎	12	13	13	12	13	13	16	13	12	26
woodwork ₍₅₀₎	13	15	15	15	15	29	34	13	9	19
zenotravel ₍₂₀₎	11	12	13	10	11	12	13	8	8	8
Sum ₍₁₆₆₇₎	696	756	760	739	758	881	994	701	664	405

Table A.3: Number of solved tasks using Cartesian abstraction heuristics.

	h^{UCP}	h^{OUCP}_{one}	h^{OUCP}_{div}	h^{GZOCP}_{one}	h^{GZOCP}_{div}	h^{SCP}_{one}	h^{SCP}_{div}	h^{CAN}	h^{PhO}	h^{OCP}
airport ₍₅₀₎	22	23	23	25	24	32	30	22	28	15
barman ₍₃₄₎	4	4	4	4	4	4	4	4	4	0
blocks ₍₃₅₎	18	21	21	18	18	28	28	26	26	8
childsnaek ₍₂₀₎	0	0	0	0	0	0	0	0	0	0
depot ₍₂₂₎	7	7	7	6	8	12	13	6	7	1
driverlog ₍₂₀₎	12	13	13	14	14	15	15	11	13	2
elevators ₍₅₀₎	33	40	40	40	42	42	44	40	42	1
floortile ₍₄₀₎	2	3	3	2	3	4	6	2	2	0
freecell ₍₈₀₎	20	20	20	22	22	61	65	25	17	0
ged ₍₂₀₎	19	19	19	15	19	15	19	15	15	0
grid ₍₅₎	2	3	3	3	3	3	3	3	3	1
gripper ₍₂₀₎	8	8	8	8	8	8	8	4	7	3
hiking ₍₂₀₎	12	13	13	12	14	13	14	13	10	2
logistics ₍₆₃₎	23	26	26	27	27	29	37	20	28	9
miconic ₍₁₅₀₎	55	71	71	70	70	87	144	42	61	20
movie ₍₃₀₎	30	30	30	30	30	30	30	30	30	30
mprime ₍₃₅₎	26	28	28	27	27	30	31	27	25	13
mystery ₍₃₀₎	18	18	18	17	18	19	19	17	17	7
nomystery ₍₂₀₎	15	19	19	19	20	20	20	17	19	2
openstacks ₍₁₀₀₎	49	49	49	49	49	51	51	12	46	7
parcprinter ₍₅₀₎	26	29	30	32	32	39	32	10	28	14
parking ₍₄₀₎	1	4	4	0	1	13	13	0	2	0
pathways ₍₃₀₎	4	4	4	4	4	4	5	4	4	4
pegsol ₍₅₀₎	44	44	44	44	46	48	48	9	44	12
pipes-nt ₍₅₀₎	18	22	22	17	19	23	24	17	16	5
pipes-t ₍₅₀₎	14	16	16	14	16	17	18	15	9	0
psr-small ₍₅₀₎	49	50	49	49	49	50	50	47	49	38
rovers ₍₄₀₎	7	8	8	8	8	8	8	5	7	5
satellite ₍₃₆₎	6	6	6	6	6	7	7	4	6	2
scanalyzer ₍₅₀₎	21	21	21	21	21	27	33	23	13	4
sokoban ₍₅₀₎	46	46	46	48	48	50	50	46	48	9
storage ₍₃₀₎	15	16	16	16	16	16	16	15	15	7
tetris ₍₁₇₎	9	9	9	9	9	11	11	4	3	0
tidybot ₍₄₀₎	23	23	23	23	23	23	23	23	22	1
tpp ₍₃₀₎	6	7	7	7	8	7	8	6	6	6
transport ₍₇₀₎	24	31	31	33	35	32	35	33	27	3
trucks ₍₃₀₎	8	10	9	8	10	13	13	5	7	3
visitall ₍₄₀₎	12	13	13	13	13	30	30	12	27	12
woodwork ₍₅₀₎	19	25	25	27	29	44	45	11	25	37
zenotravel ₍₂₀₎	12	13	13	12	13	12	13	12	13	2
Sum ₍₁₆₆₇₎	739	812	811	799	826	977	1063	637	771	285

Table A.4: Number of solved tasks using combined abstraction heuristics.

	h^{UCP}	$h_{\text{one}}^{\text{OUCP}}$	$h_{\text{one}}^{\text{GZOCP}}$	$h_{\text{one}}^{\text{SCP}}$	h^{CAN}	h^{PhO}	h^{OCP}
airport ₍₅₀₎	30	30	30	30	26	30	30
barman ₍₃₄₎	4	4	4	4	4	0	0
blocks ₍₃₅₎	27	27	26	28	27	27	26
childsnack ₍₂₀₎	0	0	0	0	0	0	0
depot ₍₂₂₎	9	9	8	9	7	7	7
driverlog ₍₂₀₎	14	14	14	14	14	14	14
elevators ₍₅₀₎	33	33	33	33	33	25	25
floortile ₍₄₀₎	2	2	2	2	2	2	2
freecell ₍₈₀₎	62	61	53	65	38	54	52
ged ₍₂₀₎	15	15	15	15	15	14	15
grid ₍₅₎	3	3	2	3	3	2	2
gripper ₍₂₀₎	7	7	7	7	7	7	6
hiking ₍₂₀₎	11	11	11	11	9	8	8
logistics ₍₆₃₎	27	27	27	27	26	25	25
miconic ₍₁₅₀₎	143	143	143	143	141	141	141
movie ₍₃₀₎	30	30	30	30	30	30	30
mprime ₍₃₅₎	21	21	21	21	21	20	20
mystery ₍₃₀₎	15	15	15	15	15	14	14
nomystery ₍₂₀₎	20	20	20	20	20	18	18
openstacks ₍₁₀₀₎	45	43	45	47	34	32	31
parcprinter ₍₅₀₎	26	26	28	28	22	26	26
parking ₍₄₀₎	6	6	0	6	4	2	1
pathways ₍₃₀₎	4	4	4	4	4	4	4
pegsol ₍₅₀₎	44	46	46	46	44	44	42
pipes-nt ₍₅₀₎	22	22	22	22	22	17	17
pipes-t ₍₅₀₎	15	13	13	14	13	10	9
psr-small ₍₅₀₎	49	49	49	49	49	49	49
rovers ₍₄₀₎	8	8	8	8	8	7	7
satellite ₍₃₆₎	7	7	7	7	7	7	7
scanalyzer ₍₅₀₎	15	15	21	23	23	17	15
sokoban ₍₅₀₎	45	45	46	46	41	40	40
storage ₍₃₀₎	16	16	16	16	16	15	15
tetris ₍₁₇₎	9	9	9	9	9	5	5
tidybot ₍₄₀₎	28	24	24	26	30	22	22
tpp ₍₃₀₎	6	6	6	6	6	6	6
transport ₍₇₀₎	25	25	25	25	23	22	22
trucks ₍₃₀₎	9	9	9	9	9	7	7
visitall ₍₄₀₎	14	14	14	29	20	28	28
woodwork ₍₅₀₎	23	23	25	25	25	23	23
zenotravel ₍₂₀₎	10	10	10	10	10	9	9
Sum ₍₁₆₆₇₎	899	892	888	932	857	830	820

Table A.5: Number of solved tasks using landmark heuristics.

	h_{div}^{SCP}				h_{LM}^{SCP}	h^{LM-cut}	h^{iPDB}	$h^{M\&S}$	h_{Sys}^{PhO}	h^{SEQ}
	HC	Sys	CART	COMB						
airport ₍₅₀₎	24	28	31	30	30	28	38	18	27	23
barman ₍₃₄₎	4	4	4	4	4	4	4	4	4	4
blocks ₍₃₅₎	28	28	28	28	28	28	28	28	26	28
childsnack ₍₂₀₎	0	0	0	0	0	0	0	0	0	0
depot ₍₂₂₎	7	12	11	13	9	7	11	7	7	7
driverlog ₍₂₀₎	14	15	14	15	14	14	13	13	13	12
elevators ₍₅₀₎	41	36	44	44	33	40	41	31	36	16
floortile ₍₄₀₎	5	5	3	6	2	13	2	6	2	6
freecell ₍₈₀₎	21	21	65	65	65	15	21	20	15	41
ged ₍₂₀₎	19	19	15	19	15	15	19	19	15	13
grid ₍₅₎	3	3	3	3	3	2	3	2	2	1
gripper ₍₂₀₎	8	8	8	8	7	7	8	20	7	7
hiking ₍₂₀₎	12	12	14	14	11	9	12	13	11	9
logistics ₍₆₃₎	27	33	39	37	27	26	31	25	26	20
miconic ₍₁₅₀₎	65	68	142	144	143	141	69	79	54	52
movie ₍₃₀₎	30	30	30	30	30	30	30	30	30	30
mprime ₍₃₅₎	24	30	27	31	21	22	24	23	21	20
mystery ₍₃₀₎	17	18	17	19	15	17	17	17	15	15
nomystery ₍₂₀₎	20	20	20	20	20	14	20	18	16	10
openstacks ₍₁₀₀₎	49	49	51	51	47	47	49	49	47	35
parcprinter ₍₅₀₎	28	38	38	32	28	33	38	45	30	48
parking ₍₄₀₎	13	13	10	13	6	6	13	6	3	6
pathways ₍₃₀₎	4	4	5	5	4	5	4	4	4	4
pegsol ₍₅₀₎	44	48	48	48	46	46	48	48	44	46
pipes-nt ₍₅₀₎	20	20	23	24	22	17	21	18	15	15
pipes-t ₍₅₀₎	16	17	16	18	14	12	18	16	9	11
psr-small ₍₅₀₎	50	50	49	50	49	49	50	50	49	50
rovers ₍₄₀₎	8	7	8	8	8	8	8	8	7	6
satellite ₍₃₆₎	6	6	7	7	7	7	6	7	6	6
scanalyzer ₍₅₀₎	27	33	23	33	23	27	23	23	11	25
sokoban ₍₅₀₎	46	50	46	50	46	50	50	47	49	35
storage ₍₃₀₎	16	16	16	16	16	15	16	15	15	15
tetris ₍₁₇₎	10	11	9	11	9	6	10	2	3	12
tidybot ₍₄₀₎	22	22	22	23	26	23	22	1	21	7
tpp ₍₃₀₎	6	6	8	8	6	7	6	8	6	8
transport ₍₇₀₎	35	24	25	35	25	23	35	24	21	21
trucks ₍₃₀₎	9	9	12	13	9	10	9	7	7	9
visitall ₍₄₀₎	28	30	16	30	29	16	28	13	27	30
woodwork ₍₅₀₎	19	49	34	45	25	29	23	32	25	23
zenotravel ₍₂₀₎	13	13	13	13	10	13	13	12	11	9
Sum ₍₁₆₆₇₎	838	905	994	1063	932	881	881	808	737	735

Table A.6: Number of solved tasks by different heuristics.

	SymBA ₂ [*]	$h_{\text{COMB}}^{\text{SCP}}$
airport ⁽⁵⁰⁾	27	41
barman ⁽³⁴⁾	16	4
blocks ⁽³⁵⁾	31	28
childsnaek ⁽²⁰⁾	4	0
depot ⁽²²⁾	7	14
driverlog ⁽²⁰⁾	14	15
elevators ⁽⁵⁰⁾	44	44
floortile ⁽⁴⁰⁾	34	16
freecell ⁽⁸⁰⁾	26	68
ged ⁽²⁰⁾	20	19
grid ⁽⁵⁾	3	3
gripper ⁽²⁰⁾	20	8
hiking ⁽²⁰⁾	20	14
logistics ⁽⁶³⁾	25	38
miconic ⁽¹⁵⁰⁾	108	142
movie ⁽³⁰⁾	30	30
mprime ⁽³⁵⁾	24	31
mystery ⁽³⁰⁾	15	19
nomystery ⁽²⁰⁾	15	20
openstacks ⁽¹⁰⁰⁾	90	51
parcprinter ⁽⁵⁰⁾	39	42
parking ⁽⁴⁰⁾	4	15
pathways ⁽³⁰⁾	5	5
pegsol ⁽⁵⁰⁾	48	48
pipesworld-notankage ⁽⁵⁰⁾	15	24
pipesworld-tankage ⁽⁵⁰⁾	16	18
psr-small ⁽⁵⁰⁾	50	50
rovers ⁽⁴⁰⁾	14	8
satellite ⁽³⁶⁾	10	7
scanalyzer ⁽⁵⁰⁾	21	33
sokoban ⁽⁵⁰⁾	48	50
storage ⁽³⁰⁾	15	16
tetris ⁽¹⁷⁾	11	12
tidybot ⁽⁴⁰⁾	27	32
tpp ⁽³⁰⁾	8	8
transport ⁽⁷⁰⁾	33	35
trucks ⁽³⁰⁾	12	15
visitall ⁽⁴⁰⁾	19	30
woodworking ⁽⁵⁰⁾	48	49
zenotravel ⁽²⁰⁾	11	13
Sum ⁽¹⁶⁶⁷⁾	1027	1115

Table A.7: Number of solved tasks with h^2 mutexes pruning by SymBA₂^{*} and $h_{\text{COMB}}^{\text{SCP}}$ (diverse saturated cost partitioning heuristics over the combination of abstraction heuristics).

BIBLIOGRAPHY

- Vidal Alcázar and Álvaro Torralba (2015). “A Reminder about the Importance of Computing and Exploiting Invariants in Planning.” In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. Ed. by Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, pp. 2–6 (cit. on p. 111).
- Christer Bäckström and Bernhard Nebel (1995). “Complexity Results for SAS⁺ Planning.” In: *Computational Intelligence* 11.4, pp. 625–655 (cit. on p. 8).
- Thomas Ball, Andreas Podelski, and Sriram K. Rajamani (2001). “Boolean and Cartesian Abstraction for Model Checking C Programs.” In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, pp. 268–283 (cit. on pp. 26, 40).
- Blai Bonet (2013). “An Admissible Heuristic for SAS⁺ Planning Obtained from the State Equation.” In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*. Ed. by Francesca Rossi. AAAI Press, pp. 2268–2274 (cit. on p. 110).
- Blai Bonet and Héctor Geffner (2001). “Planning as Heuristic Search.” In: *Artificial Intelligence* 129.1, pp. 5–33 (cit. on pp. 45, 52).
- Krishnendu Chatterjee, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar (2005). “Counterexample-guided Planning.” In: *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI 2005)*, pp. 104–111 (cit. on p. 41).
- Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith (2003). “Counterexample-Guided Abstraction Refinement for Symbolic Model Checking.” In: *Journal of the ACM* 50.5, pp. 752–794 (cit. on pp. 1, 25, 40).
- Edmund M. Clarke, Orna Grumberg, and Doron A. Peled (1999). *Model Checking*. The MIT Press (cit. on p. 40).
- Joseph C. Culberson and Jonathan Schaeffer (1998). “Pattern Databases.” In: *Computational Intelligence* 14.3, pp. 318–334 (cit. on pp. 1, 48).
- Carmel Domshlak, Malte Helmert, Erez Karpas, and Shaul Markovitch (2011). “The SelMax Planner: Online Learning for Speeding up Optimal Planning.” In: *IPC 2011 planner abstracts*, pp. 108–112 (cit. on p. 108).
- Stefan Edelkamp (2001). “Planning with Pattern Databases.” In: *Proceedings of the Sixth European Conference on Planning (ECP 2001)*. Ed. by

- Amedeo Cesta and Daniel Borrajo. AAI Press, pp. 84–90 (cit. on pp. 1, 25, 48).
- Stefan Edelkamp (2006). “Automated Creation of Pattern Database Search Heuristics.” In: *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, pp. 35–50 (cit. on pp. 16, 20).
- Gaojian Fan, Martin Müller, and Robert Holte (2017). “Additive Merge-and-Shrink Heuristics for Diverse Action Costs.” In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*. AAAI Press, pp. 4287–4293 (cit. on p. 17).
- Ariel Felner, Richard Korf, and Sarit Hanan (2004). “Additive Pattern Database Heuristics.” In: *Journal of Artificial Intelligence Research* 22, pp. 279–318 (cit. on p. 16).
- Maximilian Fickert and Jörg Hoffmann (2017). “Complete Local Search: Boosting Hill-Climbing through Online Relaxation Refinement.” In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*. AAAI Press, pp. 107–115 (cit. on p. 42).
- Maria Fox and Derek Long (2003). “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains.” In: *Journal of Artificial Intelligence Research* 20, pp. 61–124 (cit. on p. 5).
- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman (cit. on p. 23).
- Malik Ghallab, Dana Nau, and Paolo Traverso (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann (cit. on p. 1).
- Peter E. Hart, Nils J. Nilsson, and Bertram Raphael (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107 (cit. on pp. 1, 12).
- Patrik Haslum (2012). “Incremental Lower Bounds for Additive Cost Planning Problems.” In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. Ed. by Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet. AAAI Press, pp. 74–82 (cit. on pp. 12, 41).
- Patrik Haslum, Blai Bonet, and Héctor Geffner (2005). “New Admissible Heuristics for Domain-Independent Planning.” In: *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*. AAAI Press, pp. 1163–1168 (cit. on pp. 16, 20).
- Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig (2007). “Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning.” In: *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*. AAAI Press, pp. 1007–1012 (cit. on pp. 3, 16, 22, 59, 66, 81, 109).

- Patrik Haslum and Héctor Geffner (2000). "Admissible Heuristics for Optimal Planning." In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*. Ed. by Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock. AAAI Press, pp. 140–149 (cit. on p. 12).
- Malte Helmert (2006). "The Fast Downward Planning System." In: *Journal of Artificial Intelligence Research* 26, pp. 191–246 (cit. on p. 5).
- Malte Helmert and Carmel Domshlak (2009). "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*. Ed. by Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis. AAAI Press, pp. 162–169 (cit. on pp. 12, 42, 109).
- Malte Helmert, Patrik Haslum, and Jörg Hoffmann (2007). "Flexible Abstraction Heuristics for Optimal Sequential Planning." In: *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*. Ed. by Mark Boddy, Maria Fox, and Sylvie Thiébaux. AAAI Press, pp. 176–183 (cit. on pp. 13, 25).
- Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim (2014). "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces." In: *Journal of the ACM* 61.3, 16:1–63 (cit. on pp. 17, 25, 59, 110).
- Malte Helmert, Gabriele Röger, and Silvan Sievers (2015). "On the Expressive Power of Non-Linear Merge-and-Shrink Representations." In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. Ed. by Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, pp. 106–114 (cit. on p. 26).
- István T. Hernádvölgyi and Robert C. Holte (2000). "Experiments with Automatically Created Memory-Based Heuristics." In: *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA 2000)*. Ed. by Berthe Y. Choueiry and Toby Walsh. Vol. 1864. Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 281–290 (cit. on pp. 1, 26, 57).
- Jörg Hoffmann and Bernhard Nebel (2001). "The FF Planning System: Fast Plan Generation Through Heuristic Search." In: *Journal of Artificial Intelligence Research* 14, pp. 253–302 (cit. on pp. 12, 42).
- Jörg Hoffmann, Julie Porteous, and Laura Sebastia (2004). "Ordered Landmarks in Planning." In: *Journal of Artificial Intelligence Research* 22, pp. 215–278 (cit. on pp. 54, 57).
- Robert C. Holte (2013). "Korf's Conjecture and the Future of Abstraction-Based Heuristics." In: *Proceedings of the Tenth Symposium on Abstraction,*

- Reformulation, and Approximation (SARA 2013)*. Ed. by Alan M. Frisch and Peter Gregory. AAAI Press, pp. 128–131 (cit. on p. 43).
- Robert C. Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy (2006). “Maximizing over Multiple Pattern Databases Speeds up Heuristic Search.” In: *Artificial Intelligence* 170.16–17, pp. 1123–1136 (cit. on pp. 15, 43).
- Richard M. Karp (1972). “Reducibility among combinatorial problems.” In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller and James W. Thatcher. Plenum Press, pp. 85–103 (cit. on pp. 23, 84).
- Erez Karpas and Carmel Domshlak (2009). “Cost-Optimal Planning with Landmarks.” In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*. Ed. by Craig Boutilier. AAAI Press, pp. 1728–1733 (cit. on pp. 3, 13, 16, 18, 55, 108).
- Erez Karpas, Michael Katz, and Shaul Markovitch (2011). “When Optimal Is Just Not Good Enough: Learning Fast Informative Action Cost Partitionings.” In: *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*. Ed. by Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert. AAAI Press, pp. 122–129 (cit. on pp. 18, 80, 87).
- Michael Katz and Carmel Domshlak (2008). “Optimal Additive Composition of Abstraction-based Admissible Heuristics.” In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*. Ed. by Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen. AAAI Press, pp. 174–181 (cit. on pp. 2, 13, 15–17, 21).
- Michael Katz and Carmel Domshlak (2010). “Optimal admissible composition of abstraction heuristics.” In: *Artificial Intelligence* 174.12–13, pp. 767–798 (cit. on pp. 13, 16–18).
- Emil Keyder, Jörg Hoffmann, and Patrik Haslum (2012). “Semi-Relaxed Plan Heuristics.” In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. Ed. by Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet. AAAI Press, pp. 128–136 (cit. on p. 42).
- Emil Keyder, Jörg Hoffmann, and Patrik Haslum (2014). “Improving Delete Relaxation Heuristics Through Explicitly Represented Conjunctions.” In: *Journal of Artificial Intelligence Research* 50, pp. 487–533 (cit. on p. 42).
- Emil Keyder, Silvia Richter, and Malte Helmert (2010). “Sound and Complete Landmarks for And/Or Graphs.” In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*. Ed. by Helder Coelho, Rudi Studer, and Michael Wooldridge. IOS Press, pp. 335–340 (cit. on pp. 2, 54, 57).

- Richard E. Korf (1997). "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases." In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*. AAAI Press, pp. 700–705 (cit. on p. 43).
- Richard E. Korf and Ariel Felner (2002). "Disjoint Pattern Database Heuristics." In: *Artificial Intelligence* 134.1–2, pp. 9–22 (cit. on p. 16).
- Bernhard Korte and Jens Vygen (2001). *Combinatorial Optimization: Theory and Algorithms*. 2nd. Springer (cit. on p. 71).
- Levi H. S. Lelis, Santiago Franco, Marvin Abisrror, Mike Barley, Sandra Zilles, and Robert C. Holte (2016). "Heuristic Subset Selection in Classical Planning." In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. Ed. by Subbarao Kambhampati. AAAI Press, pp. 3185–3191 (cit. on p. 87).
- Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle (2010). "Iterated Local Search: Framework and Applications." In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Springer, pp. 363–397 (cit. on p. 71).
- Drew McDermott (2000). "The 1998 AI Planning Systems Competition." In: *AI Magazine* 21.2, pp. 35–55 (cit. on p. 8).
- Judea Pearl (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley (cit. on pp. 1, 11).
- David Pisinger and Stefan Ropke (2010). "Large Neighborhood Search." In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Springer, pp. 399–419 (cit. on p. 76).
- Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp (2015). "From Non-Negative to General Operator Cost Partitioning." In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*. AAAI Press, pp. 3335–3341 (cit. on pp. 10, 15, 16, 20).
- Florian Pommerening, Gabriele Röger, and Malte Helmert (2013). "Getting the Most Out of Pattern Databases for Classical Planning." In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*. Ed. by Francesca Rossi. AAAI Press, pp. 2357–2364 (cit. on pp. 3, 16, 18, 19, 23, 59, 66, 105, 110).
- Julie Porteous and Stephen Cresswell (2002). "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)*, pp. 45–54 (cit. on p. 56).
- Silvia Richter, Malte Helmert, and Matthias Westphal (2008). "Landmarks Revisited." In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*. AAAI Press, pp. 975–982 (cit. on pp. 55, 57).

- Silvia Richter and Matthias Westphal (2010). "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks." In: *Journal of Artificial Intelligence Research* 39, pp. 127–177 (cit. on p. 13).
- Stuart Russell and Peter Norvig (1995). *Artificial Intelligence — A Modern Approach*. Prentice Hall (cit. on pp. 12, 71).
- Jendrik Seipp, Florian Pommerening, and Malte Helmert (2015). "New Optimization Functions for Potential Heuristics." In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. Ed. by Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, pp. 193–201 (cit. on p. 80).
- Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert (2017). *Downward Lab*. <https://doi.org/10.5281/zenodo.790461> (cit. on p. 5).
- Silvan Sievers, Manuela Ortlieb, and Malte Helmert (2012). "Efficient Implementation of Pattern Database Heuristics for Classical Planning." In: *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*. Ed. by Daniel Borrajo, Ariel Felner, Richard Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan Sturtevant. AAAI Press, pp. 105–111 (cit. on pp. 59, 109).
- Silvan Sievers, Martin Wehrle, and Malte Helmert (2014). "Generalized Label Reduction for Merge-and-Shrink Heuristics." In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*. AAAI Press, pp. 2358–2366 (cit. on pp. 17, 48).
- Silvan Sievers, Martin Wehrle, and Malte Helmert (2016). "An Analysis of Merge Strategies for Merge-and-Shrink Heuristics." In: *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*. AAAI Press, pp. 294–298 (cit. on pp. 59, 110).
- Jan-Georg Smaus and Jörg Hoffmann (2009). "Relaxation Refinement: A New Method to Generate Heuristic Functions." In: *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MoChArt 2008)*. Ed. by Doron A. Peled and Michael J. Wooldridge, pp. 147–165 (cit. on p. 40).
- Álvaro Torralba, Carlos Linares López, and Daniel Borrajo (2016). "Abstraction Heuristics for Symbolic Bidirectional Search." In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. Ed. by Subbarao Kambhampati. AAAI Press, pp. 3272–3278 (cit. on p. 111).
- Fan Yang, Joseph Culberson, Robert Holte, Uzi Zahavi, and Ariel Felner (2008). "A General Theory of Additive State Space Abstractions." In: *Journal of Artificial Intelligence Research* 32, pp. 631–662 (cit. on p. 2).

Colophon

This thesis is typeset using the *classicthesis* package developed by André Miede (<https://bitbucket.org/amiede/classicthesis/>). The style is inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".