Journal of Artificial Intelligence Research 67 (2020) 129–167 Submitted 08/2019; published 01/2020

Saturated Cost Partitioning for Optimal Classical Planning

Jendrik Seipp JENDRIK.SEIPP@UNIBAS.CH
Thomas Keller THO.KELLER@UNIBAS.CH
Malte Helmert MALTE.HELMERT@UNIBAS.CH
University of Basel
Basel, Switzerland

Abstract

Cost partitioning is a method for admissibly combining a set of admissible heuristic estimators by distributing operator costs among the heuristics. Computing an optimal cost partitioning, i.e., the operator cost distribution that maximizes the heuristic value, is often prohibitively expensive to compute. Saturated cost partitioning is an alternative that is much faster to compute and has been shown to yield high-quality heuristics. However, its greedy nature makes it highly susceptible to the order in which the heuristics are considered. We propose a greedy algorithm to generate orders and show how to use hill-climbing search to optimize a given order. Combining both techniques leads to significantly better heuristic estimates than using the best random order that is generated in the same time. Since there is often no single order that gives good guidance on the whole state space, we use the maximum of multiple orders as a heuristic that is significantly better informed than any single-order heuristic, especially when we actively search for a set of diverse orders.

1. Introduction

A* search (Hart, Nilsson, & Raphael, 1968) with an admissible heuristic (Pearl, 1984) is one of the most prominent methods for optimal classical planning (Ghallab, Nau, & Traverso, 2004). Since a single heuristic is often unable to capture all relevant aspects of a planning task, it is desirable to combine information from multiple heuristics. One way of doing so admissibly is to maximize over multiple admissible heuristic estimates in each state (Holte, Felner, Newton, Meshulam, & Furcy, 2006). However, this method does not really *combine* multiple heuristics but merely *selects* the most informative one in each state.

Cost partitioning (Katz & Domshlak, 2008; Yang, Culberson, Holte, Zahavi, & Felner, 2008) is a more sophisticated way of combining heuristics admissibly that often produces higher estimates than any single estimator can provide. By distributing operator costs among the heuristics, cost partitioning allows to *sum* heuristic estimates admissibly. An *optimal cost partitioning* (OCP) can be computed in polynomial time for abstraction (Katz & Domshlak, 2008, 2010) and landmark (Karpas & Domshlak, 2009) heuristics. Despite these promising theoretical guarantees, it is often too expensive to compute even a single optimal cost partitioning in practice (e.g., Pommerening, Röger, & Helmert, 2013). Therefore, multiple approximations with varying time vs. accuracy tradeoffs have been proposed, such as zero-one cost partitioning (e.g., Edelkamp, 2006), uniform cost partitioning (Katz & Domshlak, 2007), the canonical heuristic (Haslum, Botea, Helmert, Bonet, & Koenig, 2007) for pattern databases, post-hoc optimization (Pommerening et al., 2013) and Lagrangian decomposition for optimal cost partitioning (Pommerening, Röger, Helmert, Cambazard, Rousseau, & Salvagnin, 2019). We refer to the literature for a theoretical and experimental comparison of cost partitioning algorithms (Seipp, Keller, & Helmert, 2017).

More recently, we introduced the *saturated cost partitioning* (SCP) algorithm (Seipp & Helmert, 2014, 2018), which exploits that operator costs can sometimes be decreased in a heuristic without affecting the quality of the heuristic. Given an ordered sequence of admissible heuristics and a cost function, the saturated cost partitioning algorithm computes all heuristic values under the given cost function. It then assigns the smallest cost function that preserves these heuristic values to the heuristic and continues the process with the next heuristic in the sequence and the remaining costs that have not been assigned to a heuristic so far.

Saturated cost partitioning assigns costs greedily and is therefore susceptible to the order in which the heuristics are considered. 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 orders that are optimized by hill-climbing search significantly improve over random orders, and we obtain the best results using a combination of both techniques. 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. Our empirical evaluation of using multiple orders shows a significant improvement over single-order heuristics. This approach is similar to the one by Karpas, Katz, and Markovitch (2011), who maximize over multiple precomputed cost partitionings that are optimized for a set of sample states. Our method has the advantage that it never computes an optimal cost partitioning, which can be prohibitively expensive even for a single state.

Finally, we show that the sets of saturated cost partitioning heuristics that are derived from multiple orders often contain heuristics that do not contribute any additional information during the search. Similarly to other work on heuristic subset selection (e.g., Lelis, Franco, Abisrror, Barley, Zilles, & Holte, 2016), we try to pick a subset of heuristics that complement each other by actively searching for multiple *diverse* orders. The resulting heuristic not only improves over using multiple non-diverse orders but also compares favorably to the state of the art for optimal classical planning.

2. Transition Systems

Cost partitioning can be applied to any collection of heuristics for state-space search, and therefore our definitions are not specific to classical planning. We first define transition systems, which are also known as state spaces.

Definition 1 (transition systems)**.** *A* transition system $\mathcal{T}$ *is a directed, labeled graph defined by a finite set of* states $S(\mathcal{T})$*, a finite set of* labels $\mathcal{L}(\mathcal{T})$*, a set* $T(\mathcal{T})$ *of labeled* transitions $s \xrightarrow{\ell} s'$ *with* $s, s' \in S(\mathcal{T})$ *and* $\ell \in \mathcal{L}(\mathcal{T})$*, an* initial state $s_0(\mathcal{T})$*, and a set* $S_\star(\mathcal{T})$ *of* goal states*.*

The objective in state-space search is to find a path from the initial state to a goal state.

Definition 2 (paths and goal paths)**.** *Let* $\mathcal{T}$ *be a transition system. A* path *from* $s \in S(\mathcal{T})$ *to* $s' \in S(\mathcal{T})$ *is a sequence of transitions from* $T(\mathcal{T})$ *of the form* $\pi = \langle s^0 \xrightarrow{\ell_1} s^1, s^1 \xrightarrow{\ell_2} s^2, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n \rangle$*, where* $s^0 = s$ *and* $s^n = s'$*. The* length *of* π*, denoted by* $|\pi|$*, is* n*. The empty path (of length 0) is permitted if* $s = s'$*.*

A goal path *from* $s \in S(\mathcal{T})$ *is a path from* s *to any goal state* $s' \in S_\star(\mathcal{T})$*. Goal paths are also called* plans*. We write* $\Pi_\star(\mathcal{T}, s)$ *for the set of goal paths from* s *in* $\mathcal{T}$*.*

So far, we have not introduced a notion of (label or path) *cost*. This does not mean that we consider the unit-cost setting but rather that cost functions must be provided *in addition to* the transition system. This separation makes it easier to consider the same transition system with varying cost functions, which is a key concept for cost partitioning.

Definition 3 (cost functions)**.** *A* cost function *for transition system $\mathcal{T}$ is a function cost : $\mathcal{L}(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$. A cost function cost is* finite *if $-\infty < cost(\ell) < \infty$ for all labels ℓ. It is* non-negative *if $cost(\ell) \geq 0$ for all labels ℓ.*

We write $\mathcal{C}(\mathcal{T})$ for the set of all cost functions for $\mathcal{T}$.

For brevity, especially in examples, we sometimes use tuple notation for cost functions, writing $cost = \langle cost(l_1), \ldots, cost(l_k) \rangle$, assuming an (arbitrary) order on the labels of the transition system. We speak of *general* cost functions when we want to emphasize that a cost function is not required to be non-negative or finite.

3. Dealing with Negative and Infinite Values

Negative costs were already considered in previous work (Pommerening, Helmert, Röger, & Seipp, 2015). However, allowing infinite costs in cost functions goes beyond previous work and is necessary to cleanly state some of our formal definitions and results. Furthermore, this extension is useful for *subset-saturated* cost partitioning (Seipp & Helmert, 2019), a variant of saturated cost partitioning that only preserves a subset of heuristic estimates. Note that in this paper, we only consider the variant that preserves all heuristic estimates.

Allowing infinities means that we must take care in arithmetic expressions that involve both $+\infty$ and $-\infty$. We will consider two different kinds of addition. *Left addition* is denoted by the regular summation operators $+$ (infix) and $\sum$ (prefix) and handles infinities as $\infty + x = \infty$ and $-\infty + x = -\infty$ for all x, including $x \in \{\infty, -\infty\}$. In particular, sums involving both kinds of infinities evaluate to the leftmost infinite value in the sum. This operation is associative, but not commutative. We will use left addition to combine multiple heuristic estimates within cost partitioning.

Path addition is denoted by the operators $\oplus$ (infix) and $\bigoplus$ (prefix) and handles infinities as $x \oplus y = \infty$ iff $x = \infty$ or $y = \infty$, and $x \oplus (-\infty) = -\infty \oplus x = -\infty$ for all $x \neq \infty$. In other words, sums involving mixed infinities evaluate to $+\infty$. This operation is associative and commutative. Path addition is used to combine the costs of multiple transitions along a path. We will interpret transitions of cost $-\infty$ as "infinitely cheap" and transitions of cost ∞ as "non-existent", which explains why $-\infty \oplus \infty = \infty$: a path that uses a non-existent transition cannot really be used and therefore has infinite cost even if it uses an infinitely cheap transition. Of course, this is just an intuitive interpretation. The formal reason for these definitions is that they allow the usual theorems on properties of heuristics and cost partitioning to generalize to cases involving infinite costs.

With finite values, both kinds of summation follow the usual rules of addition. Note that both operations behave identically when at least one of the two operands is finite. In fact, in sums involving two operands, the only difference is that $-\infty + \infty = -\infty$, while $-\infty \oplus \infty = \infty$.

4. Weighted Transition Systems

By combining transition systems and cost functions, we obtain *weighted* transition systems.

Definition 4 (weighted transition systems)**.** *A* weighted transition system *is a pair $\langle \mathcal{T}, \mathit{cost} \rangle$ where $\mathcal{T}$ is a transition system and cost is a cost function for $\mathcal{T}$.*

As usual, we extend cost functions from labels to paths.

Definition 5 (cost of a path)**.** *The* cost *of a path $\pi = \langle s^0 \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n \rangle$ in a weighted transition system $\langle \mathcal{T}, \mathit{cost} \rangle$ is defined as $\mathit{cost}(\pi) = \bigoplus_{i=1}^{n} \mathit{cost}(\ell_i)$.*

Note that by our definition of path addition ($\bigoplus$), the cost of any path including a label of cost ∞ is ∞, even if the path also includes labels of cost $-\infty$.

We can now define the notion of optimal paths.

Definition 6 (goal distances and optimal paths)**.** *The* goal distance *of a state $s \in S(\mathcal{T})$ in a weighted transition system $\langle \mathcal{T}, \mathit{cost} \rangle$ is defined as $\inf_{\pi \in \Pi_\star(\mathcal{T}, s)} \mathit{cost}(\pi)$, where $\Pi_\star(\mathcal{T}, s)$ is the set of goal paths from s in $\mathcal{T}$. (The infimum of the empty set is ∞.)*

We write $h^*_{\mathcal{T}}(\mathit{cost}, s)$ for the goal distance of s in $\langle \mathcal{T}, \mathit{cost} \rangle$ and omit $\mathcal{T}$ from the notation where the transition system does not matter or is clear from context.

A goal path π from s is optimal *under the given cost function if $\mathit{cost}(\pi) = h^*_{\mathcal{T}}(\mathit{cost}, s)$.*

We define the goal distances as an infimum rather than a minimum because it is possible that no minimum exists if there are negative-cost cycles in the transition system.

In general, we have $h^*(\mathit{cost}, s) \in \mathbb{R} \cup \{-\infty, \infty\}$, so goal distances can be negative or infinite. If *cost* is non-negative, then so is h^*. However, finite *cost* does not imply finite h^*: $h^*(\mathit{cost}, s) = -\infty$ can follow from cycles with negative finite cost and $h^*(\mathit{cost}, s) = \infty$ from lack of goal paths.

In *optimal classical planning*, we are given a compact description of a transition system and a finite non-negative cost function, and the objective is to find an optimal goal path for the initial state or show that no such goal path exists.

5. Heuristics

Heuristics are functions that estimate the distance of a given state to a goal state (Pearl, 1984). The literature usually defines heuristics as functions only of states, i.e., for a fixed cost function. We define them as functions of cost functions and states so that we can introduce the notion of cost partitioning cleanly.

Definition 7 (heuristics, goal-awareness, admissibility and consistency)**.** *A* heuristic *for a transition system $\mathcal{T}$ is a function $h : \mathcal{C}(\mathcal{T}) \times S(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$.*

- *Heuristic h is* goal-aware *if $h(\mathit{cost}, s) \leq 0$ for all cost functions $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and goal states $s \in S_\star(\mathcal{T})$.*
- *Heuristic h is* admissible *if $h(\mathit{cost}, s) \leq h^*_{\mathcal{T}}(\mathit{cost}, s)$ for all cost functions $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and all states $s \in S(\mathcal{T})$.*
- *Heuristic h is* consistent *if $h(\mathit{cost}, s) \leq \mathit{cost}(\ell) \oplus h(\mathit{cost}, s')$ for all cost functions $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and all transitions $s \xrightarrow{\ell} s' \in T(\mathcal{T})$.*

Heuristics are used in *heuristic search algorithms* like A* (Hart et al., 1968) or greedy best-first search (Doran & Michie, 1966) to find a path from the initial state to a goal. Greedy best-first search does not guarantee optimality of solutions, and A* requires an admissible heuristic to guarantee that solutions are optimal. Consistency is usually desirable for A* to avoid the extra work that is caused by *reopening* of states. It is well-known that if only finite non-negative cost functions are permitted, goal-aware and consistent heuristics are admissible (Russell & Norvig, 1995). In Appendix A.1, we show that this result continues to hold for general cost functions.

6. Abstraction Heuristics

In all cases where we consider specific heuristics in this paper, these are *abstraction heuristics* (e.g., Edelkamp, 2001; Helmert, Haslum, & Hoffmann, 2007; Katz & Domshlak, 2008, 2010). An abstraction heuristic for a transition system $\mathcal{T}$ is defined by a transition system $\mathcal{T}'$ called the *abstract transition system* and a function $\alpha : S(\mathcal{T}) \to S(\mathcal{T}')$ called the *abstraction function*. The abstraction mapping must preserve goal states and transitions; we refer to the literature for details (Helmert et al., 2007). Heuristic values are computed by mapping states of $\mathcal{T}$ (*concrete states*) to states of $\mathcal{T}'$ (*abstract states*) and computing the goal distance in the abstract transition system: $h(\mathit{cost}, s) = h^*_{\mathcal{T}'}(\mathit{cost}, \alpha(s))$. We introduced goal distances for general cost functions in Definition 6, so abstraction heuristics for general cost functions are well-defined. Abstraction heuristics for finite non-negative cost functions are admissible and consistent (e.g., Helmert et al., 2007). We show in Appendix A that admissibility and consistency generalize to arbitrary cost functions.

Note, however, that generalizing *implementations* of abstraction heuristics is more challenging than generalizing their definition. For non-negative cost functions, abstract goal distances can be computed with Dijkstra's algorithm (Dijkstra, 1959), which can be implemented with worst-case runtime $O(N \log N + M)$, where $N = |S(\mathcal{T}')|$ and $M = |T(\mathcal{T}')|$ (Cormen, Leiserson, & Rivest, 1990). For possibly negative cost functions, no algorithms are known that substantially outperform the Bellman-Ford algorithm (Bellman, 1958) in the worst case, whose worst-case runtime is $O(NM)$. If we consider a (not particularly large) abstract transition system with $N = 10000$ states and at least $N \log N$ transitions, this means that the Bellman-Ford algorithm is 10000 times slower than Dijkstra's algorithm (ignoring the constant factors hidden in the big-O notation).

General cost functions give rise to another issue: it is well-known that using an admissible heuristic in an A* search results in optimal solutions if only non-negative cost functions are allowed (Hart et al., 1968). This result does not hold for general cost functions since A* (like Dijkstra's algorithm) cannot handle negative cost-cycles.

Neither issue is problematic for our work, however. Even though we allow general cost functions in cost partitionings, we only need to compute abstract goal distances under non-negative cost functions (see Appendix D). As a consequence, all goal distances are non-negative, which in turn ensures that all cost-partitioned heuristic estimates are non-negative (see Appendix E). Therefore, we can use Dijkstra's algorithm to compute abstract goal distances and A* to find optimal plans.

7. 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 to com-

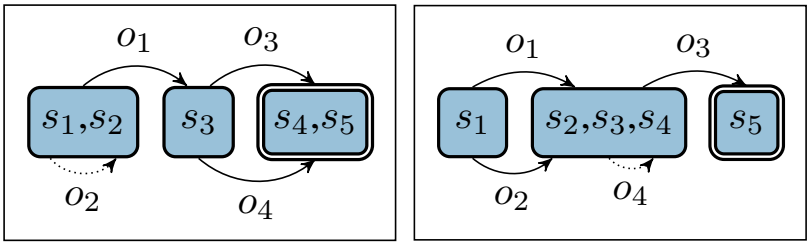


Figure 1: Example abstraction heuristics. The cost function is $cost = \langle 4, 1, 4, 1 \rangle$, i.e., operators o_1 and o_3 cost 4, whereas o_2 and o_4 cost 1. In all figures depicting abstraction heuristics, circles depict concrete states, rounded rectangles are abstract states and dotted arrows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

bine admissible heuristics so that the resulting heuristic is both informative and remains 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 & Domshlak, 2008, 2010). Cost partitioning is more general than maximizing over multiple heuristics, as the cost partitioning that assigns all costs to the heuristic with the highest estimate yields the same result as a heuristic that maximizes over all component heuristics.

Definition 8 (cost partitioning)**.** *Let $\mathcal{T}$ be a transition system. A* cost partitioning *for a cost function $cost \in \mathcal{C}(\mathcal{T})$ is a tuple $\langle cost_1, \ldots, cost_n \rangle \in \mathcal{C}(\mathcal{T})^n$ whose sum is bounded by cost: $\sum_{i=1}^{n} cost_i(\ell) \leq cost(\ell)$ for all $\ell \in \mathcal{L}(\mathcal{T})$.*

A cost partitioning algorithm *for a transition system $\mathcal{T}$ and a tuple of heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ takes a cost function cost as its input and produces a cost partitioning $\langle cost_1, \ldots, cost_n \rangle$ for cost as its output. It induces the* cost-partitioned heuristic $h(cost, s) = \sum_{i=1}^{n} h_i(cost_i, s)$.

Cost-partitioned heuristics derived from admissible (consistent) component heuristics are admissible (consistent). Cost partitioning forms the basis of most state-of-the-art heuristics in optimal classical planning (e.g., Karpas & Domshlak, 2009; Helmert & Domshlak, 2009; Pommerening et al., 2013; Seipp & Helmert, 2014; Pommerening et al., 2015; Seipp et al., 2017; Seipp, 2017).

Katz and Domshlak (2008) studied cost partitioning in the case where the overall cost function *cost* and component cost functions $cost_i$ are finite and non-negative. Pommerening et al. (2015) generalized this by allowing negative costs in $cost_i$ (but not in *cost*). We show in Appendices B and C that the consistency and admissibility results also apply to our more general definition.

Example 1. *We illustrate the concept of cost partitioning with the two abstraction heuristics h_1 and h_2 in Figure 1 and the cost function $cost = \langle 4, 1, 4, 1 \rangle$. We have $h_1(cost, s_1) = 5$ and $h_2(cost, s_1) = 5$. Therefore, maximizing over the two estimates also yields a heuristic value of 5. With a suitable cost partitioning we can obtain a more accurate heuristic estimate. 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 the admissible estimate $h^{\mathcal{C}}(cost, s_1) = 5 + 3 = 8$.*

It is possible to compute an optimal cost partitioning for a given state (i.e., a cost partitioning resulting in the largest possible heuristic value for the given state) in polynomial time for abstraction (Katz & Domshlak, 2008, 2010) and landmark (Karpas & Domshlak, 2009) heuristics. These results were initially proved for non-negative cost functions, later generalized to possibly negative cost functions (Pommerening et al., 2015), and it is not difficult to further generalize them to handle infinite costs.

Definition 9 (optimal cost partitioning)**.** *Let $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ be a tuple of heuristics for a weighted transition system $\langle \mathcal{T}, cost \rangle$. A cost partitioning $\mathcal{C}^*$ is* optimal *for cost, $\mathcal{H}$ and a state $s \in S(\mathcal{T})$ if $h^{\mathcal{C}^*}(s)$ is maximal among all cost partitionings for cost and $\mathcal{H}$.*

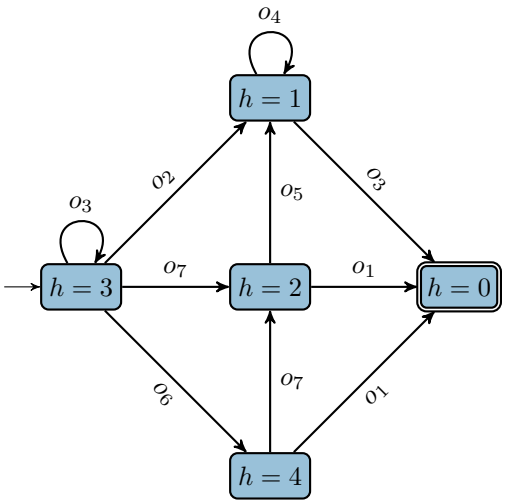
Example 2. *For the abstraction heuristics h_1 and h_2 in Figure 1 the example cost partitioning $\mathcal{C}$ from Example 1 is an optimal cost partitioning: $\mathcal{C} = \langle cost_1, cost_2 \rangle$ with $cost_1 = \langle 4, 0, 1, 1 \rangle$ and $cost_2 = \langle 0, 0, 3, 0 \rangle$ yields $h^{\mathcal{C}}(cost, s_1) = 5 + 3 = 8$. However, there are multiple optimal cost partitionings: for example, the cost partitioning $\mathcal{C}' = \langle cost_1, cost_2 \rangle$ with $cost_1 = \langle 3.5, 0, 1, 1 \rangle$ and $cost_2 = \langle 0.5, 1, 3, 0 \rangle$ also yields $h^{\mathcal{C}'}(cost, s_1) = 4.5 + 3.5 = 8$.*

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 & Domshlak, 2009; Katz & Domshlak, 2010; Karpas et al., 2011). 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). We give further evidence for the impractical time and memory requirements of optimal cost partitioning in Section 13.1.

8. Saturated Cost Partitioning

Saturated cost partitioning (Seipp & Helmert, 2014, 2018) is a greedy algorithm that quickly computes suboptimal cost partitionings. It has been shown to perform significantly better experimentally than other techniques such as optimal cost partitioning, uniform cost partitioning, greedy zero-one cost partitioning, and the canonical heuristic for pattern databases (Seipp et al., 2017). Saturated cost partitioning is based on the observation that parts of the costs are "wasted" on a component heuristic if they do not contribute to its heuristic estimate. 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 $\mathcal{T}'$ associated with abstraction heuristic h in Figure 2. The abstract states in $\mathcal{T}'$ are labeled with their goal distances ($h = X$). The cost function *cost* is depicted in the table to the right of Figure 2. It assigns o_1 a cost of 5, but no goal distance changes if the cost of o_1 is reduced to 4. Consequently, a part of the cost for o_1 can be "saved" and used for a different heuristic. The cost of o_4, which only occurs as a self loop in the transition system, can even be reduced to 0 without affecting any goal distance. Similarly, decreasing the cost of operator o_6 to -1 does not change any goal distance. In a subsequent heuristic, o_1 can therefore have a cost of $5 - 4 = 1$, o_4 can have a cost of 4 and o_6 can have a cost of $7 - (-1) = 8$.



o	$cost(o)$	$mscf(o)$
o_1	5	$\max(4-0, 2-0) = 4$
o_2	2	$\max(3-1) = 2$
o_3	1	$\max(3-3, 1-0) = 1$
o_4	4	$\max(1-1) = 0$
o_5	1	$\max(2-1) = 1$
o_6	7	$\max(3-4) = -1$
o_7	2	$\max(3-2, 4-2) = 2$

Figure 2: (Figure 8 in Seipp & Helmert, 2018) Left: abstract transition system of an example planning task. Every transition is labeled with an operator. Right: costs and minimum saturated costs that suffice to preserve all goal distances in the abstract transition system.

8.1 Saturated Cost Functions

The insight that we can use a lower cost function without changing any heuristic values is captured formally by the concept of *saturated cost functions*. 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 $cost(\ell)$.

Definition 10 (saturated cost function)**.** *Consider a transition system $\mathcal{T}$, a heuristic h for $\mathcal{T}$ and a cost function $cost \in \mathcal{C}(\mathcal{T})$. A cost function $scf \in \mathcal{C}(\mathcal{T})$ is* saturated *for h and cost if*

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

Note that in addition to these two requirements the original definition of saturated cost functions (Seipp & Helmert, 2014) required a saturated cost function to be minimal. We drop this requirement since for some classes of heuristics (e.g., the $h^{\max}$ heuristic by Bonet & Geffner, 2001) it is not guaranteed that there is a unique minimum. Using Definition 10, 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*.[1]

Obviously, only saturated cost functions *scf* with $scf(l) < cost(\ell)$ 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 useful saturated cost function for a given heuristic and cost function depends on the type of heuristic. We call functions that perform this computation *saturators*.

Definition 11 (saturators)**.** *Consider a transition system $\mathcal{T}$ and a heuristic h for $\mathcal{T}$.*

A saturator *for h is a function $saturate : \mathcal{C}(\mathcal{T}) \to \mathcal{C}(\mathcal{T})$ such that $saturate(cost)$ is a saturated cost function for h and cost.*

1. We can also define saturated cost functions for a subset of states in a transition system (Seipp & Helmert, 2019), but in this work we only consider saturated cost functions that preserve the heuristic values of all states.

Saturators have not previously been introduced in the literature: earlier work exploited results on the uniqueness of minimal saturated cost functions for abstraction heuristics to simply speak of "the" minimal saturated cost function. We introduce the concept of saturators here to generalize the definition of saturated cost partitioning to heuristics without a unique minimum saturated cost function.

Saturated cost partitioning considers an ordered sequence of heuristics.

Definition 12 (heuristic orders)**.** *Let $\mathcal{H}$ be a finite non-empty set of heuristics. An* order *ω of $\mathcal{H}$ is a sequence of heuristics that contains each heuristic in $\mathcal{H}$ exactly once. We write $\Omega(\mathcal{H})$ for the set of all orders of $\mathcal{H}$.*

We can now formally define saturated cost partitioning. The main differences to earlier definitions (e.g., Seipp & Helmert, 2018) are that we consider general cost functions and parameterize the definition by saturators.

Definition 13 (saturated cost partitioning)**.** *Consider a weighted transition system $\langle \mathcal{T}, \mathit{cost} \rangle$, a heuristic order $\omega = \langle h_1, \ldots, h_n \rangle \in \Omega(\mathcal{H})$ for $\mathcal{T}$ and a sequence $\langle \mathit{saturate}_1, \ldots, \mathit{saturate}_n \rangle$ such that $\mathit{saturate}_i$ is a saturator for h_i for all $1 \leq i \leq n$.*

The saturated cost partitioning *$\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle$ of the cost function cost for order ω induced by the sequence of saturators $\langle \mathit{saturate}_1, \ldots, \mathit{saturate}_n \rangle$ is defined as:*

$$
\begin{aligned}
\mathit{remain}_0 &= \mathit{cost} \\
\mathit{cost}_i &= \mathit{saturate}_i(\mathit{remain}_{i-1}) && \text{for all } 1 \leq i \leq n \\
\mathit{remain}_i &= \mathit{remain}_{i-1} - \mathit{cost}_i && \text{for all } 1 \leq i \leq n,
\end{aligned}
$$

where the auxiliary cost functions remain_i represent the remaining costs after processing the first i heuristics in ω.

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

The subtraction in the definition of remain_i follows the rules of left addition and the definition $a - b := a + (-b)$. Hence, if $\mathit{remain}_{i-1}(\ell)$ is (positively or negatively) infinite, then we always obtain $\mathit{remain}_i(\ell) = \mathit{remain}_{i-1}(\ell)$. In particular, when the leftover costs for a label are ∞, we may allocate cost ∞ to all further cost functions because $\infty - \infty = \infty$ under left addition.

It is easy to see that the saturated cost partitioning is indeed a cost partitioning (Definition 8), i.e., $\sum_{i=1}^{n} \mathit{cost}_i(\ell) \leq \mathit{cost}(\ell)$ for all labels ℓ. For labels ℓ with $\mathit{cost}(\ell) = \infty$ this holds trivially. For labels ℓ with $\mathit{cost}(\ell) = -\infty$, we must have $\mathit{remain}_i(\ell) = -\infty$ for all $0 \leq i \leq n$ because $-\infty - x = -\infty$ for all $x \in \mathbb{R} \cup \{-\infty, \infty\}$. Hence we get $\mathit{cost}_i(\ell) = -\infty$ for all $1 \leq i \leq n$ because cost_i is bounded by remain_{i-1} by the definition of saturated cost functions. With $n \geq 1$, this shows $\sum_{i=1}^{n} \mathit{cost}_i(\ell) = -\infty = \mathit{cost}(\ell)$.

It remains to consider the case where $\mathit{cost}(\ell)$ is finite. If all $\mathit{cost}_i(\ell)$ are finite or $-\infty$, the cost partitioning property is easy to show, so consider the case where $\mathit{cost}_i(\ell) = \infty$ for some $1 \leq i \leq n$. Let i_0 be the smallest index with this property. Then we must have $\mathit{remain}_{i_0-1}(\ell) = \infty$. With finite $\mathit{cost}(\ell)$, this is only possible if $\mathit{cost}_j(\ell) = -\infty$ for some $j < i_0$, which implies $\sum_{i=1}^{n} \mathit{cost}_i(\ell) = -\infty < \mathit{cost}(\ell)$ due to the rules of left addition (there must be a $-\infty$ before ∞ in the sum).

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. However, it is an open research question which properties a heuristic must have for it to possess a saturator that always returns a minimum saturated cost function.

8.2 Minimum Saturated Cost Function for Abstraction Heuristics

We showed in previous work that for an abstraction heuristic h and a cost function *cost* there is always a saturator that computes the unique minimum saturated cost function *mscf* for h and *cost* (Seipp & Helmert, 2018). The key idea is to make sure that for each label ℓ, the consistency constraint $h(\mathit{mscf}, s) \leq \mathit{mscf}(\ell) + h(\mathit{mscf}, s')$ is tight for at least one state transition $s \xrightarrow{\ell} s'$. This can be enforced by setting

$$\mathit{mscf}(\ell) = \sup_{a \xrightarrow{\ell} b \in T(\mathcal{T}')} (h^*_{\mathcal{T}'}(\mathit{cost}, a) \ominus h^*_{\mathcal{T}'}(\mathit{cost}, b)), \tag{1}$$

where $\mathcal{T}'$ is the abstract transition system underlying h. The $\ominus$ operator behaves like regular subtraction for finite values and handles infinities as $x \ominus y = -\infty$ iff $x = -\infty$ or $y = \infty$, and $x \ominus y = \infty$ iff $x = \infty \neq y$ or $x \neq -\infty = y$. We prove in Appendix F that *mscf* is the minimum saturated cost function even for general cost functions.

The minimum saturated cost function can be computed with negligible overhead during the construction of pattern databases (Culberson & Schaeffer, 1998; Edelkamp, 2001), Cartesian abstractions (Ball, Podelski, & Rajamani, 2001; Seipp & Helmert, 2013) and merge-and-shrink abstractions not using *label reduction* (Sievers, Wehrle, & Helmert, 2014).[2]

Figure 2 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, 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.

For all abstraction heuristics h and cost functions *cost*, we use Equation 1 to obtain a saturator *saturate-abstraction*$_h$ which returns the minimum saturated cost function for h and *cost*.

Example 3. *We use the two abstraction heuristics h_1 and h_2 from Figure 1 to show a complete run of the saturated cost partitioning algorithm. We choose the order $\langle h_1, h_2 \rangle$ and the saturators $\langle$saturate-abstraction$_{h_1}$, saturate-abstraction$_{h_2}\rangle$. 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-abstraction$_{h_1}$(remain$_0$) $= \langle 4, 0, 1, 1 \rangle$*

2. 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 1 Interleaved saturated cost partitioning algorithm. Given a weighted transition system, it computes a set of abstraction heuristics and corresponding minimum saturated cost functions that form a cost partitioning.

```
1: procedure INTERLEAVEDSATURATEDCOSTPARTITIONING(⟨T, cost⟩)
2:     while not TERMINATIONCONDITION do
3:         T' ← compute abstraction of T using cost
4:         h ← abstraction heuristic corresponding to T'
5:         mscf ← minimum saturated cost function for h and cost
6:         cost ← cost − mscf
```

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-abstraction$_{h_2}$(remain$_1$) $= \langle 0, 0, 3, 0\rangle$. The two saturated cost functions form a cost partitioning and we have $h^{SCP}_{\langle h_1,h_2\rangle}(cost, s_1) = 5 + 3 = 8$. 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.

9. Implementation and Evaluation Details

Before we describe the main contributions of this paper, we briefly discuss some details concerning the implementation and evaluation of our work.

9.1 Interleaved Saturated Cost Partitioning Algorithm

In earlier work (Seipp & Helmert, 2018), we exploited 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 1 shows in pseudo-code how this approach computes abstraction heuristics for a weighted transition system $\langle \mathcal{T}, cost\rangle$ that are cost-partitioned with saturated cost partitioning: starting with *cost*, it iteratively creates an abstract transition system $\mathcal{T}'$ using *cost*, obtains the corresponding abstraction heuristic h, computes the minimum saturated cost function *mscf* for h and *cost*, subtracts *mscf* from *cost* and proceeds to the next iteration with the remaining costs.

The procedure terminates 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.

In this procedure, we compute the abstractions "just-in-time", guiding the computation of abstractions by the current remaining cost function. In the present work, 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 our original approach, we fix the set of heuristics before computing saturated cost partitionings. We never encountered memory problems in our experimental evaluation because of the fact that we have to maintain all abstractions in memory until the search starts.

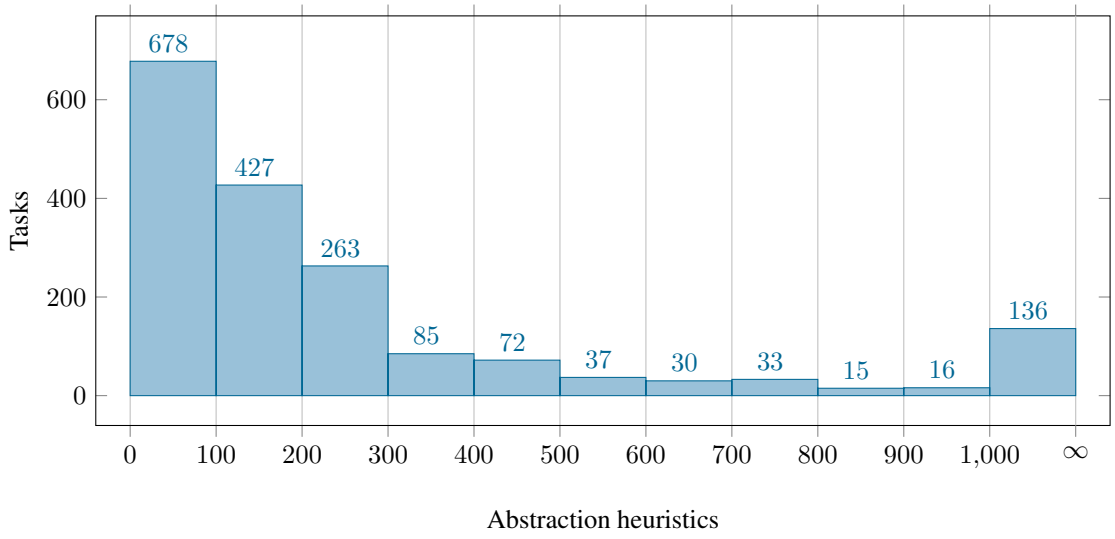


Figure 3: Number of abstraction heuristics computed for the tasks from our benchmark set.

9.2 Storing Only Useful Lookup Tables

As is common, we make the evaluation of an abstraction heuristic h with abstract transition system $\mathcal{T}'$ efficient by precomputing all goal distances in $\mathcal{T}'$ under the given cost function *cost* and storing them in a *lookup table* before the A* search starts. If we compute saturated cost partitioning heuristics for n abstraction heuristics and m orders, we need to store $n \cdot m$ lookup tables. To reduce the memory usage and the time needed for looking up goal distances in the tables, we only store *useful* lookup tables.

A lookup table is useful if it contains at least one positive finite goal distance. To see this, first note that all distances in the lookup tables are non-negative, since the remaining cost function is always non-negative. Second, if a lookup table contains only 0-values, we can ignore the table without changing any heuristic estimates. Finally, if the original cost function is finite and non-negative, the remaining cost function only assigns ∞ to a label ℓ if ℓ can never be on a goal path. Therefore, whether a heuristic h yields $h(\mathit{cost}, s) = \infty$ does not depend on *cost*. Consequently, we can store infinite estimates once for each abstraction, instead of once per lookup table.

9.3 Experimental Setup

Since the techniques we introduce build upon one another, we interleave theoretical and experimental analyses. Before starting with the first analysis, we briefly describe our setup for all experiments in the paper. We use all 1827 tasks without conditional effects from the optimization tracks of the 1998–2018 International Planning Competitions (IPC) and limit time to 30 minutes and memory to 3.5 GiB. As our set of heuristics we use the combination of pattern databases found by hill climbing in the space of pattern collections (Haslum et al., 2007), systematic pattern databases of sizes 1 and 2 (Pommerening et al., 2013) and Cartesian abstractions of *landmark* and *goal* task decompositions (Seipp & Helmert, 2018). We choose the combination of these heterogeneous heuristics instead of a homogeneous subset, because having more (and different) heuristics makes ordering them more difficult, which increases the impact of the ordering algorithms and therefore makes their evaluation

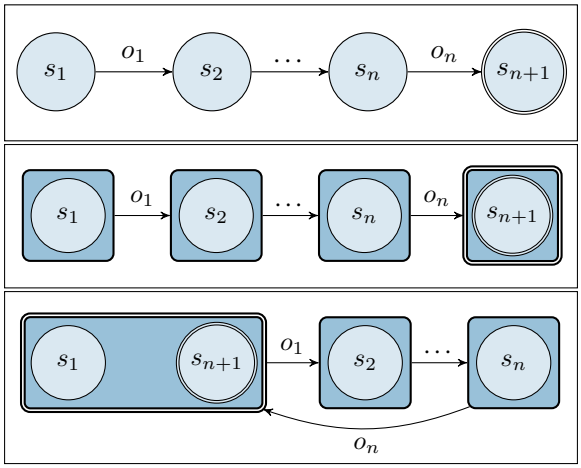


Figure 4: Concrete transition system $\mathcal{T}$ (top) with abstraction heuristics h_1 (middle) and h_2 (bottom). All labels have cost 1.

easier. Figure 3 shows that the number of heuristics varies a lot between tasks. For 678 tasks we compute at most 100 heuristics, but there is also one task for which we obtain more than 45 000 heuristics.

All algorithms are implemented in the Fast Downward planning system (Helmert, 2006) and we use the Downward Lab toolkit (Seipp, Pommerening, Sievers, & Helmert, 2017) to conduct experiments. All benchmarks[3], code[4] and experimental data[5] have been published online.

10. Single 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 lead to different heuristic estimates for the same state.

Theorem 1 (importance of good orders)**.** *There exist weighted transition systems $\langle \mathcal{T}, cost \rangle$, sets of heuristics $\mathcal{H}$ for $\mathcal{T}$ and states $s \in S(\mathcal{T})$ such that $h^{SCP}_{\omega}(cost, s) > h^{SCP}_{\omega'}(cost, s)$ for two orders $\omega, \omega' \in \Omega(\mathcal{H})$.*

Proof. Consider the example in Figure 1. For the two abstraction heuristics h_1 and h_2, we have $h^{\text{SCP}}_{\langle h_1, h_2 \rangle}(s_2, cost) = 8 > h^{\text{SCP}}_{\langle h_2, h_1 \rangle}(s_2, cost) = 7$. □

Note that we can easily enlarge the accuracy gap between two heuristics resulting from two different orders. For the example in Figure 1, it suffices to raise the cost of labels o_1 and o_2 simultaneously to increase the difference between $h^{\text{SCP}}_{\langle h_1, h_2 \rangle}(cost, s_2)$ and $h^{\text{SCP}}_{\langle h_2, h_1 \rangle}(cost, s_2)$. Similar instances can also be constructed for unit-cost tasks, and Figure 4 shows an example: for the concrete transition system $\langle \mathcal{T}, cost \rangle$ with $cost(\ell) = 1$ for all labels $\ell \in \mathcal{L}(\mathcal{T})$ and the two abstraction heuristics h_1 and h_2 for $\mathcal{T}$, we have $h^{\text{SCP}}_{\langle h_1, h_2 \rangle}(cost, s_1) = n$ and $h^{\text{SCP}}_{\langle h_2, h_1 \rangle}(cost, s_1) = 0$. Note that this

3. Benchmarks: `https://doi.org/10.5281/zenodo.2616479`
4. Code: `https://doi.org/10.5281/zenodo.3497367`
5. Experimental data: `https://doi.org/10.5281/zenodo.3497396`

example also works if the abstraction underlying h_1 does not map each concrete state to a different abstract state.

Although Theorem 1 highlights the importance of choosing good orders, previous work on saturated cost partitioning (Seipp & Helmert, 2014) only considered two non-random ordering methods based on the h^{add} heuristic (Bonet & Geffner, 2001). Seipp and Helmert showed experimentally that neither order consistently outperforms the other nor the random order. Due to this disappointing result and the fact that both orders only work for the landmark and goal task decompositions by Seipp and Helmert (2014) and not for other abstraction heuristics, we do not investigate these orders further and present more general approaches for finding good orders instead.

There are two challenges when trying to find good orders for saturated cost partitioning: 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 weighted transition system $\langle \mathcal{T}, \mathit{cost} \rangle$, 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 $h^{\text{SCP}}_{\omega}(s)$. 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 & Norvig, 1995), to actively *search* in the space of orders.

10.1 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 and many problems allow finding a good initial solution *greedily* (e.g., Korte & 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 14 (heuristic scoring functions)**.** *Let $\mathcal{T}$ be a 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 $q : \mathcal{H} \times \mathcal{C}(\mathcal{T}) \times S(\mathcal{T}) \to \mathbb{R}$.*

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

We define three scoring functions

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

where

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

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 $q_{\frac{h}{stolen}}$ 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 $saturate_h(cost)(\ell)$ form the part of $cost(\ell)$ that h *wants*. Then $free(h, cost, \ell)$ 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., $stolen(h, cost, \ell)$, mainly depends on $free(h, cost, \ell)$, i.e., the part of the label cost that no other heuristic wants. If the amount of free costs is non-negative, h steals the costs it wants minus the costs no other heuristic wants. If there are more free costs than h wants, it steals no costs. If all costs are wanted by the other heuristics, the second case applies. When h wants costs ≥ 0, the stolen costs equal the wanted costs, since $free(h, cost, \ell) \leq 0$. Otherwise, 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

Algorithm 2 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(H, cost, s, q)
2:     ω ← ⟨⟩
3:     while H ≠ ∅ do
4:         h ← arg max_{h'∈H} q(h', cost, s)
5:         ω ← ω ⊕ ⟨h⟩
6:         H ← H \ {h}
7:         cost ← cost − saturate_h(cost)
8:     return ω
```

	random	h	*stolen*	$\frac{h}{\mathit{stolen}}$
random	–	390	548	108
h	**1068**	–	**717**	291
stolen	**892**	495	–	83
$\frac{h}{\mathit{stolen}}$	**1347**	**677**	**923**	–

Table 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. The results for *random* are averaged over 10 runs.

costs for other heuristics that want them. Table 1 holds several examples that show how to compute $\mathit{stolen}(h, \mathit{cost}, \ell)$.

10.1.1 DYNAMIC GREEDY ORDERS

We can plug any of the scoring functions into Algorithm 2 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 randomly. Results from an experiment not reported here in detail show that breaking ties with a second scoring function has a negligible influence on the quality of the resulting heuristic.

10.1.2 COMPARISON OF SCORING FUNCTIONS

We evaluate Algorithm 2 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 10 random orders. Table 2 shows a pairwise comparison of the algorithms in terms of the number of tasks where one heuristic computes a higher heuristic estimate for the initial state than the other. The heuristic estimate for *random* corresponds to the average over the 10 random orders. We see that q_h (maximizing heuristic values) and q_{stolen} (minimizing the sum of stolen costs) usually yield better

Algorithm 3 Static greedy ordering algorithm. Given a set of admissible heuristics $\mathcal{H}$, a cost function *cost*, a state s and a scoring function q, it sorts the heuristics by their respective scores in descending order.

1: **function** STATICGREEDYORDER($\mathcal{H}$, *cost*, s, q)
2: **return** $\langle h_1, \ldots, h_n \rangle \in \Omega(\mathcal{H})$ with $q(h_i, \mathit{cost}, s) \geq q(h_{i+1}, \mathit{cost}, s)$ for $1 \leq i < n$

orders than random orders for the initial state, but there are also many tasks where using random orders is preferable to using q_h or q_{stolen}. No clear conclusion can be drawn from comparing q_h to q_{stolen}: q_h compares better to random orders, while q_{stolen} compares favorably in a direct comparison between the two orders. 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 $q_{\frac{h}{stolen}}$ in subsequent experiments.

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.

10.1.3 STATIC GREEDY ORDERS

By removing line 7 from Algorithm 2 we obtain a *static* version of the ordering algorithm. It precomputes the heuristic estimate and saturated cost function for each heuristic under the original cost function once. Since the algorithm does not iteratively update the cost function, we can rewrite it without using a while loop as shown in Algorithm 3.

We compare dynamic and static greedy orders empirically by computing both orders for the initial state of each task in our benchmark set. The dynamic version yields an order that results in a higher heuristic estimate for the initial state for 539 tasks, while the opposite is the case for only 124 tasks.

As noted above, the higher heuristic values come at a price though. Figure 5 shows that static greedy orders are found much faster than dynamic greedy orders for almost all evaluated tasks, often by a margin of several orders of magnitude. There are 9 tasks for which both methods run out of memory while computing an order. For all of the remaining tasks we can compute a static greedy order in less than 100 seconds. In contrast, there are 47 tasks for which we fail to compute a dynamic greedy order in 30 minutes.

The slow computation time for dynamic orders is also a problem when we run an A* search using the saturated cost partitioning heuristics computed for the initial state. While the static variant solves 1013 tasks, the dynamic version only solves 1002 tasks.

Since static orders are much faster to compute and lead to solving more tasks than dynamic orders, we only consider static orders in subsequent experiments and often refer to them simply as "greedy orders".

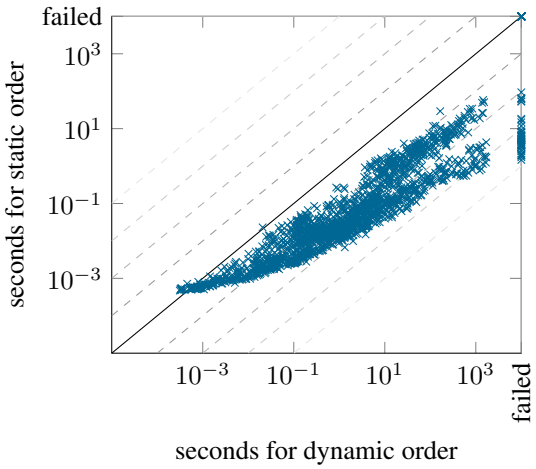


Figure 5: Time in seconds for computing a single dynamic greedy and static greedy order and the corresponding saturated cost partitioning. Each cross corresponds to a task from the benchmark set.

10.2 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 orders by switching any two positions in ω. More precisely, we switch positions 1 and 2, 1 and 3, ..., 1 and n, 2 and 3, 2 and 4, ..., $n-1$ and n. This *two-exchange* neighborhood is common for local search optimization algorithms (Pisinger & Ropke, 2010) and guarantees that all orders can be reached from any initial order. The first neighbor ω' with $h^{\text{SCP}}_{\omega'}(cost, s) > h^{\text{SCP}}_{\omega}(cost, s)$ becomes the new incumbent. We repeat this procedure until no neighbor is better than the incumbent or until a timeout 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 4. *Figure 6 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 $\langle h_2, h_3, h_1 \rangle$ 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 ($h^{SCP}_{\langle h_1, h_3, h_2 \rangle}(cost, s) = 7$) than the incumbent order, so we make $\langle h_1, h_3, h_2 \rangle$ the new incumbent.*[6] *In the next round, the first neighbor $\langle h_3, h_1, h_2 \rangle$ becomes the new incumbent.*

6. 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.

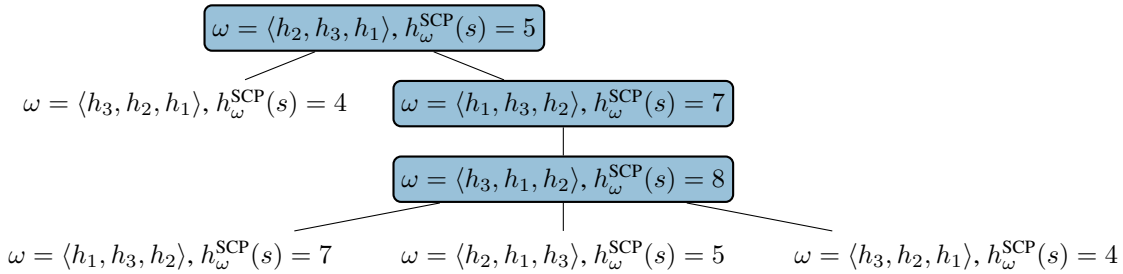


Figure 6: Example run of the hill climbing algorithm optimizing the order of three heuristics. We highlight the incumbent order in each iteration of the hill climbing algorithm.

Time limit X	0s	10s	100s	1000s	1500s
max-random-Xs	943	1006	**1014**	1013	1005
random-opt-Xs	943	980	1007	**1012**	1003
greedy-opt-Xs	1013	1031	1037	**1041**	1035

Table 3: Number of solved tasks by saturated cost partitioning using different ordering algorithms and optimization time limits. The time limits include the time for computing the initial greedy order.

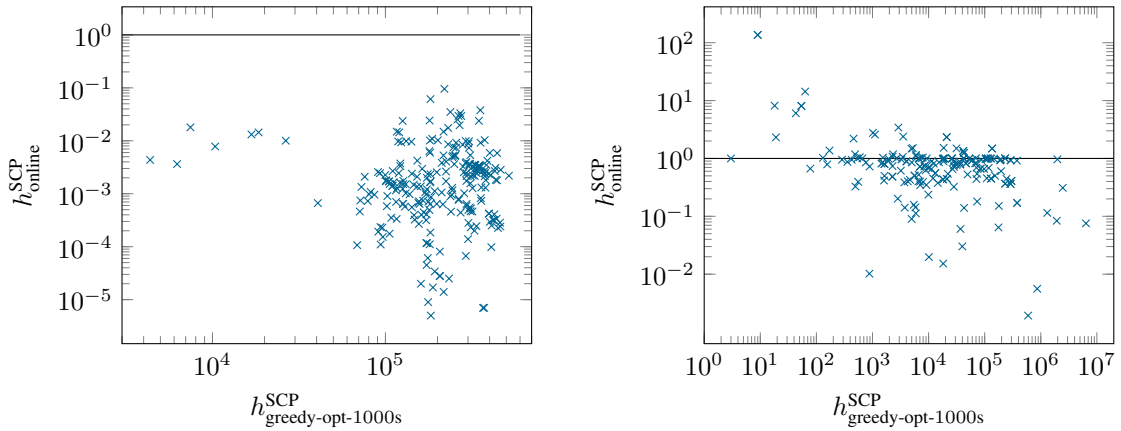
Afterwards, none of the neighbors improves upon the incumbent, so we abort the procedure and return the incumbent $\langle h_3, h_1, h_2 \rangle$.

Table 3 compares the number of solved tasks for a random (*random-opt-Xs*) and (static) greedy (*greedy-opt-Xs*) initial order that is optimized with the proposed simple hill climbing algorithm using different time limits. In addition, the table holds results for a simple baseline (*max-random-Xs*) that repeatedly computes random orders for X seconds and returns the generated order with the highest estimate for the initial state. The purpose of the baseline is to ensure that it is the combination of initial order and search that leads to improved heuristic estimates and not the large amount of orders that is considered in the available time.

All three methods benefit from a larger time limit and, as expected, the variants based on random orders profit more from it than the greedy orders. Total coverage increases from 943 tasks to 1007–1014 tasks for *max-random-Xs* and *random-opt-Xs* when going from a single unoptimized random order to 100–1000 seconds of optimization. Optimizing greedy orders incurs a smaller increase in coverage since a single greedy order already solves 70 more tasks than a single random order and is already on par with the best optimized random orders. Still, the best variant in this experiment optimizes a greedy order for 1000 seconds and solves 1041 tasks, an improvement of 28 tasks compared to a single unoptimized greedy order. This shows that the best orders can be obtained by starting with a greedy order and optimizing it afterwards.

11. 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.



(a) Evaluations per second.

(b) Expansions before the last f layer.

Figure 7: Comparison of $h^{\text{SCP}}_{\text{greedy-opt-1000s}}$ and $h^{\text{SCP}}_{\text{online}}$. Each $\langle x, y \rangle$ point corresponds to a task for which $h^{\text{SCP}}_{\text{greedy-opt-1000s}}$ returns a value of x and $h^{\text{SCP}}_{\text{online}}$ returns $x \cdot y$. Therefore, points below $y = 1$ correspond to tasks where $h^{\text{SCP}}_{\text{online}}$ yields a lower value than $h^{\text{SCP}}_{\text{greedy-opt-1000s}}$. We exclude tasks for which any of the two algorithms needs less than 1000 evaluations. Note that all axes use a log scale.

Theorem 2 (multiple states need multiple orders)**.** *There exist weighted transition systems $\langle \mathcal{T}, cost \rangle$, sets of heuristics $\mathcal{H}$ for $\mathcal{T}$ and states $s, s' \in S(\mathcal{T})$ such that $h^{SCP}_{\omega}(cost, s) > h^{SCP}_{\omega'}(cost, s)$, and $h^{SCP}_{\omega}(cost, s') < h^{SCP}_{\omega'}(cost, s')$ for two orders $\omega \neq \omega' \in \Omega(\mathcal{H})$.*

Proof. Consider the abstraction heuristics h_1 and h_2 and the cost function *cost* in Figure 1 on page 134. We have $h^{\text{SCP}}_{\langle h_1, h_2 \rangle}(cost, s_2) = 8$, $h^{\text{SCP}}_{\langle h_2, h_1 \rangle}(cost, s_2) = 7$, $h^{\text{SCP}}_{\langle h_1, h_2 \rangle}(cost, s_4) = 3$, and $h^{\text{SCP}}_{\langle h_2, h_1 \rangle}(cost, s_4) = 4$. □

Theorem 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 greedy order and the corresponding saturated cost partitioning in every evaluated state. (We hypothesize that computing a saturated cost partitioning for every evaluated state is quite expensive by itself, so we do not spend additional time to optimize the greedy orders.)

The resulting heuristic, called $h^{\text{SCP}}_{\text{online}}$, solves 709 tasks in total, 332 fewer tasks than the best saturated cost partitioning heuristic we have seen so far, $h^{\text{SCP}}_{\text{greedy-opt-1000s}}$. The difference in coverage stems from the fact that computing a saturated cost partitioning indeed slows down the evaluation significantly. Figure 7a compares the number of evaluations per second between $h^{\text{SCP}}_{\text{greedy-opt-1000s}}$ and $h^{\text{SCP}}_{\text{online}}$. 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 7b, but this is not enough to compensate for the reduced evaluation speed.

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

Our findings for $h^{\text{SCP}}_{\text{online}}$ are in line with other results from the literature which already contains many examples where increased heuristic accuracy does not compensate for the additional compu-

Orders N	1	10	100	1000
N-greedy	1013	1114	**1120**	1023
N-greedy-opt-10s	1031	1134	**1136**	–
N-greedy-opt-100s	1037	**1141**	–	–
N-greedy-opt-1000s	**1041**	–	–	–

Table 4: Number of solved tasks when maximizing over saturated cost partitioning heuristics for N unoptimized and optimized greedy orders.

tation time spent at every evaluated state (e.g., Karpas et al., 2011; Seipp, Pommerening, & Helmert, 2015).

12. Multiple Orders

Since Section 11 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.

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 $h^{\text{SCP}}_{\text{greedy}}$ to estimate the plan cost and to avoid using samples s with $h^{\text{SCP}}_{\text{greedy}}(\mathit{cost}, s) = \infty$. A slight modification of the sampling procedure turned out to have a noticeable effect in our 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.

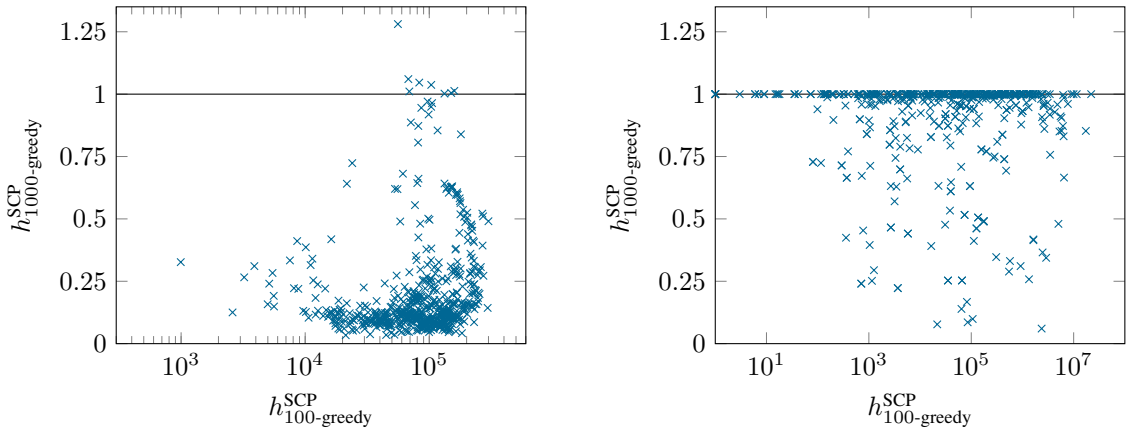
The optimization time limit does not include the time for computing the final cost partitioning. This extra time is negligible if we only compute a single order, but can become relevant if N is large and the task contains many large abstractions. We therefore only consider configurations in the following where the total time spent computing orders and cost partitionings can be limited to at most 1000 seconds (e.g., we consider a configuration where 1 order is optimized for 1000 seconds as well as one where 10 orders are optimized for 100 seconds each, but not the configuration where 10 orders are optimized for 1000 seconds).

12.1 Evaluation of Using Multiple Orders

Table 4 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.

12.1.1 NUMBER OF ORDERS

First, we investigate the impact of changing the number of orders (x axis in Table 4). The number of solved tasks increases from 1013 to 1120 when going from 1 to 100 unoptimized greedy orders. This striking difference in coverage of 107 tasks underlines the importance of Theorem 2: we need



(a) Evaluations per second.

(b) Expansions before last f layer.

Figure 8: Comparison of 100 and 1000 saturated cost partitioning heuristics using greedy orders. Each $\langle x, y \rangle$ point corresponds to a task for which $h^{\text{SCP}}_{\text{100-greedy}}$ has a value of x and $h^{\text{SCP}}_{\text{1000-greedy}}$ has a value of $x \cdot y$. Therefore, points below $y = 1$ correspond to tasks where $h^{\text{SCP}}_{\text{100-greedy}}$ has a higher value than $h^{\text{SCP}}_{\text{1000-greedy}}$. 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.

multiple orders to cover multiple states. Coverage decreases to a value close to the coverage score of a single greedy order if 1000 instead of 100 orders are used. See Section 12.1.4 for an in-depth analysis of this result. The results are similar for optimized greedy orders: using more than one optimized order is highly beneficial for all tested optimization time limits.

12.1.2 OPTIMIZATION

Next, we look at the influence of optimization on the quality of the resulting heuristics (y axis in Table 4). The first column repeats the values from the *greedy-opt-Xs* row in Table 3. As we saw there, optimizing a single order increases the coverage score. This is also true for the optimization of multiple orders, however the difference in coverage is smaller. This is no surprise, since multiple unoptimized orders already solve many more tasks than a single order. Starting with 10 greedy orders and optimizing them for 10 and 100 seconds raises the number of solved tasks from 1114 to 1134 and 1141, respectively.

12.1.3 OPTIMIZATION VS. NUMBER OF ORDERS

Last, we inspect how the number of orders and the optimization time limit interact. The results show that using multiple orders is much more important than optimizing them. For example, if we want to use 100 seconds for computing optimized greedy orders, we can optimize 1 order for 100 seconds and solve 1037 tasks, or 10 orders for 10 seconds and solve 1134 tasks. Overall, the configuration with the highest total coverage (of 1141 tasks) uses 10 greedy orders that are optimized for 100 seconds each.

12.1.4 ACCURACY VS. EVALUATION TIME

We saw above that using 1000 instead of 100 greedy orders leads to solving fewer tasks. Since adding an order to an existing set of orders can only increase the accuracy of the resulting heuristic, the gain in accuracy from including additional orders must be outweighed by the increased computational and memory cost. To test this hypothesis, we compare $h^{\text{SCP}}_{\text{100-greedy}}$ and $h^{\text{SCP}}_{\text{1000-greedy}}$ in more detail. For the 101 tasks solved by $h^{\text{SCP}}_{\text{100-greedy}}$ but not by $h^{\text{SCP}}_{\text{1000-greedy}}$, $h^{\text{SCP}}_{\text{1000-greedy}}$ runs out of memory while computing cost partitionings in 53 cases and it runs out of memory during the search in 1 case. For the remaining 47 tasks, the A* search runs out of time. We see the reason for this last finding in Figure 8. The left plot (Figure 8a) shows the number of evaluations $h^{\text{SCP}}_{\text{1000-greedy}}$ makes per second, relative to the number of evaluations per second by $h^{\text{SCP}}_{\text{100-greedy}}$. The evaluation speed drops visibly for the vast majority of tasks when multiplying the number of optimized greedy orders by 10. The evaluation speed of $h^{\text{SCP}}_{\text{1000-greedy}}$ drops below 75% of the speed of $h^{\text{SCP}}_{\text{100-greedy}}$ for 577 of the 595 commonly solved tasks with at least 1000 evaluations. One might expect that evaluating $h^{\text{SCP}}_{\text{1000-greedy}}$ takes roughly ten times as long as evaluating $h^{\text{SCP}}_{\text{100-greedy}}$ for all tasks, but often 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 8b reveals that the number of expansions excluding the last f layer remains the same or roughly the same for the majority of tasks and only for 75 of the 595 commonly solved tasks with at least 1000 evaluations the number of expansions decreases by more than 75%. Together, the two plots in Figure 8 show that indeed the increase in accuracy does not compensate for the additional evaluation time.

12.2 Probably Useful Orders

This analysis suggests that many orders do not contribute to the overall heuristic once the set of orders Ω reaches a certain size. 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*[7] 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 useless orders evaluates exactly the same states to the same heuristic values as a search that considers all orders.

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 $\langle \omega_1, \ldots, \omega_n \rangle$ (any order suffices). Let $\langle \mathcal{T}, \mathit{cost} \rangle$ be a weighted 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., $h^{\text{SCP}}_{\omega_i}(\mathit{cost}, s) > h^{\text{SCP}}_{\omega_j}(\mathit{cost}, s)$ for all $j < i$, and $h^{\text{SCP}}_{\omega_i}(\mathit{cost}, s) \geq h^{\text{SCP}}_{\omega_j}(\mathit{cost}, s)$ 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.

Figure 9 compares the percentage of probably useful orders for two different heuristics. For the moment, we are only interested in the $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$ heuristic on the x-axis, which shows the percentage of probably useful orders out of 10 optimized greedy orders. As we can see, even one of the strongest saturated cost partitioning heuristics we have described so far, $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$, contains many useless orders. In only 25% of the commonly solved tasks at least 80% of the orders

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

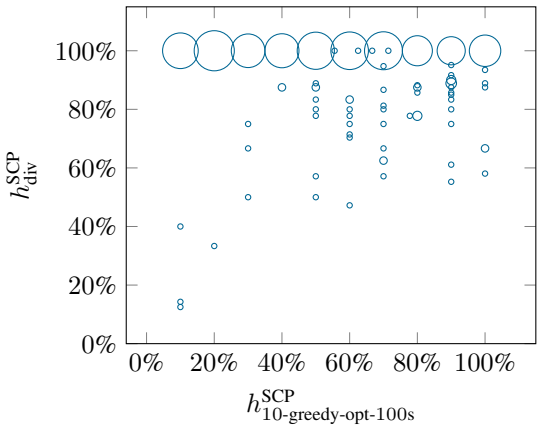


Figure 9: Percentage of *probably useful* orders for 10 optimized greedy orders ($h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$) and diverse orders with a diversification time limit of 1000 seconds ($h^{\text{SCP}}_{\text{div}}$). We exclude tasks for which any of the two heuristics uses fewer than 1000 expansions. The area of each circle is proportional to the number of tasks that it represents. Note that $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$ uses fewer than 10 orders for 5 tasks because the total preprocessing time is limited by 1000 seconds and computing the final cost partitionings takes too long.

are probably useful. If we take into account that the number of probably useful orders is 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.

12.3 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 generate a set $\hat{S}$ of 1000 sample states with the previously described sampling procedure from Haslum et al. (2007), which serves as a proxy for the real set of states encountered during the search and is used to decide if an order is probably useful or not.

12.3.1 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 $h^{\text{SCP}}_{\omega}(\mathit{cost}, s) > \max_{\omega' \in \Omega} h^{\text{SCP}}_{\omega'}(\mathit{cost}, s)$ 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

Diversification time	0s	1s	10s	20s	50s	100s	200s	500s	1000s	1200s	1500s
Coverage	1013	1038	1121	1128	1132	1134	**1136**	**1136**	1133	1128	1105

Table 5: Number of solved tasks by diverse saturated cost partitioning heuristics for unoptimized 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-10s	opt-100s	opt-1000s	Coverage
opt-0s	–	1	1	3	**13**	1133
opt-1s	**4**	–	1	5	**15**	1137
opt-10s	**5**	**3**	–	**5**	**14**	1142
opt-100s	**7**	**6**	**5**	–	**14**	**1145**
opt-1000s	3	1	1	0	–	1098

Table 6: Coverage comparison of saturated cost partitioning heuristics diversified for 1000 seconds using different optimization time limits. Each (x, y) cell holds the number of domains in which configuration y solves more tasks than configuration x. We highlight the maximum of the entries in (x, y) and (y, x) in bold. Right: Total number of solved tasks.

by the diversification procedure above. For example, when we only kept the orders that were part of a greedily-approximated minimal hitting set for $\hat{S}$, 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.

12.3.2 NUMBER OF DIVERSE ORDERS

During the diversification process we do not impose any limit on the number of orders kept 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 a simple experiment 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.

12.3.3 EVALUATION OF DIVERSE ORDERS

We evaluate the diversification procedure using unoptimized greedy orders. Table 5 shows the total number of tasks solved by the resulting heuristic for various diversification time limits T. It solves more tasks with increasing T until it reaches the peak of 1136 solved tasks at $T = 200$ and $T = 500$ seconds. Afterwards, coverage decreases again.

We saw in Table 4 that the best configuration using multiple unoptimized greedy orders solves 1120 tasks. By diversifying unoptimized greedy orders we are able to raise the total coverage by up to 16 tasks, showing that diversifying the set of orders is useful.

In the next experiment, we evaluate whether *optimized* greedy orders also benefit from diversification. We use a fixed diversification time limit of 1000 seconds, but vary the time for hill climbing in the space of orders. Table 6 holds a per-domain coverage comparison and total coverage results. The setting that yields the strongest heuristic uses 100 seconds of hill-climbing optimization. We use the name $h^{\text{SCP}}_{\text{div}}$ for this configuration (1000 seconds of diversification and at most 100 seconds of optimization for each greedy order). It solves 1145 of the 1827 tasks in our benchmark set, 4 tasks more than $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$, the best configuration using a fixed number of non-diverse greedy orders optimized for 100 seconds. In addition, $h^{\text{SCP}}_{\text{div}}$ has the advantage over $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$ that we do not need to find a good value for the number of orders when given a new task.

We believe that one of the reasons for $h^{\text{SCP}}_{\text{div}}$ solving more tasks than $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$ is that the selected orders are more diverse, and hence there are fewer relevant states where the heuristic guidance of $h^{\text{SCP}}_{\text{div}}$ is poor. This is true even though the average size of Ω is lower for $h^{\text{SCP}}_{\text{div}}$ (arithmetic mean: 6.9 orders, standard deviation: 7.87), compared to 10 optimized greedy orders. For 170 tasks, only a single order is chosen, and there is only one task for which more than 100 orders are selected during diversification (107 orders). 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: the percentage of probably useful orders of $h^{\text{SCP}}_{\text{div}}$ is higher than for $h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$ in almost all tasks. Moreover, for 98% of the analyzed tasks at least 60% of the orders of $h^{\text{SCP}}_{\text{div}}$ are probably useful, and there is even a large percentage of tasks (87%) where almost all orders of $h^{\text{SCP}}_{\text{div}}$ (at least 95%) are probably useful.

12.4 Summary of Improvements

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 7a shows per-domain and total coverage results for the heuristics that correspond to these improvements. Using a greedy order instead of a random one and using multiple orders instead of a single one led to the biggest changes in total coverage: 86.7 and 100 additionally solved tasks, respectively. In comparison, optimization and diversification were responsible for smaller differences in coverage and led to solving 28 and 4 additional tasks, respectively. These 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, $h^{\text{SCP}}_{\text{div}}$, is a huge improvement over the saturated cost partitioning heuristic we started with, $h^{\text{SCP}}_{\text{random}}$. $h^{\text{SCP}}_{\text{div}}$ solves as many or more tasks than $h^{\text{SCP}}_{\text{random}}$ in all domains and raises the total coverage score by 218.7 tasks. This increase in coverage is remarkable, since task difficulty tends to scale exponentially in optimal classical planning.

13. Comparison to Other Approaches

The strong results for $h^{\text{SCP}}_{\text{div}}$ raise the question how close it approximates the optimal cost partitioning heuristic and how it compares to other admissible heuristics and state-of-the-art optimal planners.

	$h^{\text{SCP}}_{\text{random}}$	$h^{\text{SCP}}_{\text{greedy}}$	$h^{\text{SCP}}_{\text{greedy-opt-1000s}}$	$h^{\text{SCP}}_{\text{10-greedy-opt-100s}}$	$h^{\text{SCP}}_{\text{div}}$
agricola (20)	**0.0**	**0**	**0**	**0**	**0**
airport (50)	24.3	24	24	**25**	**25**
barman (34)	**4.0**	**4**	**4**	**4**	**4**
blocks (35)	26.7	**28**	**28**	**28**	**28**
childsnack (20)	**0.0**	**0**	**0**	**0**	**0**
data-network (20)	13.0	13	**14**	**14**	**14**
depot (22)	11.4	11	12	**13**	**13**
driverlog (20)	12.9	14	**15**	**15**	**15**
elevators (50)	34.9	37	42	44	**45**
floortile (40)	3.8	**6**	**6**	**6**	**6**
freecell (80)	36.6	65	65	**66**	**66**
ged (20)	15.7	15	15	**19**	**19**
grid (5)	2.6	**3**	**3**	**3**	**3**
gripper (20)	**8.0**	**8**	**8**	**8**	**8**
hiking (20)	13.0	13	13	**14**	**14**
logistics (63)	25.4	27	29	36	**39**
miconic (150)	85.4	91	91	143	**144**
movie (30)	**30.0**	**30**	**30**	**30**	**30**
mprime (35)	26.2	28	29	**30**	**30**
mystery (30)	18.0	18	**19**	**19**	**19**
nomystery (20)	16.7	**20**	**20**	**20**	**20**
openstacks (100)	49.4	**51**	**51**	**51**	**51**
organic (20)	**7.0**	**7**	**7**	**7**	**7**
organic-split (20)	**10.0**	**10**	**10**	**10**	**10**
parcprinter (50)	26.0	30	30	**39**	**39**
parking (40)	12.4	13	13	**14**	**14**
pathways (30)	4.0	4	4	**5**	**5**
pegsol (50)	46.4	**48**	**48**	**48**	**48**
petri-net (20)	**0.0**	**0**	**0**	**0**	**0**
pipes-nt (50)	21.2	22	23	**25**	24
pipes-t (50)	15.2	16	**17**	**17**	**17**
psr-small (50)	**50.0**	**50**	**50**	**50**	**50**
rovers (40)	**8.0**	**8**	**8**	**8**	**8**
satellite (36)	6.0	**7**	**7**	**7**	**7**
scanalyzer (50)	23.2	23	27	**33**	**33**
snake (20)	12.3	**13**	11	12	**13**
sokoban (50)	**50.0**	**50**	**50**	**50**	**50**
spider (20)	14.2	**15**	**15**	**15**	**15**
storage (30)	**16.0**	**16**	**16**	**16**	**16**
termes (20)	11.5	12	12	**13**	12
tetris (17)	**11.0**	**11**	**11**	**11**	**11**
tidybot (40)	23.6	**25**	**25**	**25**	**25**
tpp (30)	6.6	7	7	**8**	**8**
transport (70)	29.4	32	33	**35**	**35**
trucks (30)	10.6	12	**13**	**13**	**13**
visitall (40)	13.0	**30**	**30**	**30**	**30**
woodwork (50)	28.4	34	44	**49**	**49**
zenotravel (20)	12.3	12	12	**13**	**13**
Sum (1827)	926.3	1013	1041	1141	**1145**

(a) A* search with variants of h^{SCP}.

	Comp1	Comp2	PPDBs	Scorpion
agricola (20)	**6**	**6**	**6**	2
airport (50)	21	23	23	**37**
barman (34)	**11**	**11**	**11**	4
blocks (35)	**30**	**30**	**30**	28
childsnack (20)	0	1	**4**	0
data-network (20)	13	13	13	**14**
depot (22)	7	7	7	**14**
driverlog (20)	14	13	14	**15**
elevators (50)	37	37	37	**44**
floortile (40)	**34**	**34**	**34**	16
freecell (80)	22	26	27	**70**
ged (20)	16	**19**	**19**	**19**
grid (5)	2	2	2	**3**
gripper (20)	**20**	**20**	**20**	8
hiking (20)	15	**18**	**18**	13
logistics (63)	26	28	28	**36**
miconic (150)	105	104	104	**143**
movie (30)	**30**	**30**	**30**	**30**
mprime (35)	19	21	20	**31**
mystery (30)	13	15	16	**19**
nomystery (20)	12	18	18	**20**
openstacks (100)	**74**	**74**	**74**	51
organic (20)	**7**	**7**	**7**	**7**
organic-split (20)	**13**	**13**	**13**	**13**
parcprinter (50)	38	41	40	**50**
parking (40)	2	2	5	**15**
pathways (30)	**5**	4	4	**5**
pegsol (50)	48	48	48	**50**
petri-net (20)	16	**18**	**18**	0
pipes-nt (50)	15	20	18	**25**
pipes-t (50)	13	16	16	**18**
psr-small (50)	**50**	**50**	**50**	**50**
rovers (40)	**14**	13	13	11
satellite (36)	**11**	10	**11**	9
scanalyzer (50)	21	21	23	**33**
snake (20)	10	12	11	**13**
sokoban (50)	47	48	48	**50**
spider (20)	10	11	11	**16**
storage (30)	15	15	15	**16**
termes (20)	14	**15**	**15**	13
tetris (17)	10	12	11	**13**
tidybot (40)	24	29	29	**32**
tpp (30)	9	**14**	8	8
transport (70)	26	28	28	**35**
trucks (30)	11	10	10	**15**
visitall (40)	16	20	20	**30**
woodwork (50)	45	46	46	**50**
zenotravel (20)	**13**	**13**	**13**	**13**
Sum (1827)	1030	1086	1086	**1207**

(b) IPC 2018 optimal planners.

Table 7: Number of solved tasks by different algorithms. We average over 10 runs for $h^{\text{SCP}}_{\text{random}}$.

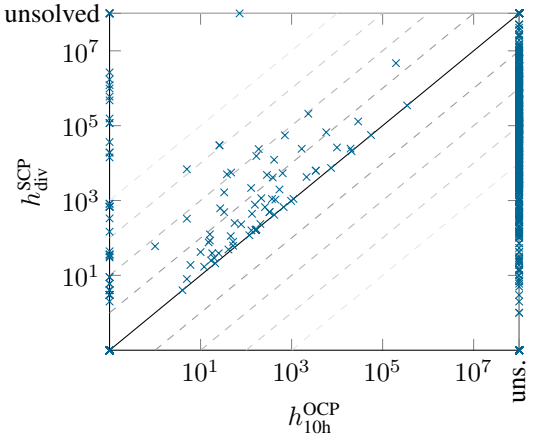


Figure 10: Number of expansions before the last f layer for $h^{\text{OCP}}_{\text{10h}}$ and $h^{\text{SCP}}_{\text{div}}$.

13.1 Optimal Cost Partitioning

We compare $h^{\text{SCP}}_{\text{div}}$ against the algorithm that computes an optimal cost partitioning in every state evaluated during the search (h^{OCP}). Since $h^{\text{SCP}}_{\text{div}}$ is much faster to evaluate than h^{OCP}, $h^{\text{SCP}}_{\text{div}}$ solves significantly more tasks than h^{OCP}. While $h^{\text{SCP}}_{\text{div}}$ solves 1145 tasks in 30 minutes, h^{OCP} finds a solution for only 291 tasks in the same amount of time. Even if we raise the time limit for optimal cost partitioning to 10 hours ($h^{\text{OCP}}_{\text{10h}}$), it solves only 374 tasks.

Figure 10 compares the number of expansions needed by $h^{\text{SCP}}_{\text{div}}$ and $h^{\text{OCP}}_{\text{10h}}$. The 1827 benchmark tasks can be divided into the following groups: 665 tasks are solved by neither heuristic. For 7 tasks both heuristics detect unsolvability and for 4 tasks only $h^{\text{OCP}}_{\text{10h}}$ is able to do so. There are 777 tasks solved by $h^{\text{SCP}}_{\text{div}}$ which $h^{\text{OCP}}_{\text{10h}}$ fails to solve and for 6 tasks the opposite is the case. For 253 commonly solved tasks, both heuristics are perfect, i.e., they need no expansions before the last f layer. There are 42 commonly solved tasks for which $h^{\text{OCP}}_{\text{10h}}$ is perfect, but $h^{\text{SCP}}_{\text{div}}$ is not and 73 commonly solved tasks for which neither heuristic is perfect.

For the 777 tasks solved by $h^{\text{SCP}}_{\text{div}}$ but not by $h^{\text{OCP}}_{\text{10h}}$, $h^{\text{OCP}}_{\text{10h}}$ runs out of time and memory while computing the optimal cost partitioning for the initial state in 215 and 314 cases, respectively. For the remaining 248 tasks $h^{\text{OCP}}_{\text{10h}}$ runs out of time during the A* search.

$h^{\text{SCP}}_{\text{div}}$ needs more expansions than $h^{\text{OCP}}_{\text{10h}}$ in 104 of the 368 commonly solved tasks, revealing that $h^{\text{SCP}}_{\text{div}}$ is often not as accurate as h^{OCP}. This shows that one can still hope to find better approximations of optimal cost partitionings. Pommerening et al. (2019) recently introduced an anytime approximation algorithm that eventually converges to an optimal cost partitioning. Their algorithm can be seeded with any cost partitioning and they showed that it is able to increase the heuristic value of saturated cost partitioning heuristics. However, so far their method has not been used as part of a planning algorithm and it remains to be seen whether or not the increase in heuristic accuracy outweighs the additional computation cost.

13.2 Other Admissible Heuristics

We refer to the literature for an experimental comparison between saturated cost partitioning and other cost partitioning algorithms (Seipp et al., 2017). In the following, we compare $h^{\text{SCP}}_{\text{div}}$ to some

	$h^{\text{SCP}}_{\text{div}}$	h^{BJOLP}	$h^{\text{iPDB}}_{\text{900s}}$	$h^{\text{LM-cut}}$	$h^{\text{SEQ+}}_{\text{LM-cut}}$	h^{pot}	$h^{\text{M\&S}}$	h^{SEQ}
Coverage	**1145**	979	960	958	957	937	866	784
#Domains $h^{\text{SCP}}_{\text{div}}$ better	–	30	24	32	31	30	29	34
#Domains $h^{\text{SCP}}_{\text{div}}$ worse	–	4	5	5	7	6	3	3

Table 8: Overall and per-domain comparison of $h^{\text{SCP}}_{\text{div}}$ to some of the strongest admissible heuristics from the literature.

of the strongest admissible heuristics from the literature, namely the h^{BJOLP} landmark heuristic (Domshlak, Helmert, Karpas, & Markovitch, 2011), h^{iPDB} (Haslum et al., 2007), $h^{\text{LM-cut}}$ (Helmert & Domshlak, 2009), the operator counting heuristic with the state equation and LM-Cut constraints $h^{\text{SEQ+}}_{\text{LM-cut}}$ (Pommerening, Röger, Helmert, & Bonet, 2014), the diverse potentials heuristic h^{pot} (Seipp et al., 2015), merge-and-shrink ($h^{\text{M\&S}}$) using bisimulation and the SCC-DFP merge strategy (Helmert, Haslum, Hoffmann, & Nissim, 2014; Sievers, Wehrle, & Helmert, 2016), and the state-equation heuristic h^{SEQ} (Bonet, 2013). The approach by Karpas et al. (2011), which precomputes multiple optimal cost partitionings, is inapplicable in our setting, as there are already 604 tasks solved by $h^{\text{SCP}}_{\text{div}}$ for which we fail to compute even a single optimal cost partitioning within 30 minutes and 3.5 GiB.

Table 8 shows the total coverage scores and also compares the algorithms to $h^{\text{SCP}}_{\text{div}}$ on a per-domain basis. The $h^{\text{SCP}}_{\text{div}}$ heuristic significantly outperforms all other heuristics, solving 166 more tasks than the heuristic with the second-highest coverage score in this comparison, h^{BJOLP}. Beyond total coverage, $h^{\text{SCP}}_{\text{div}}$ also has an edge over the other heuristics in most individual domains. For example, out of the 48 tested domains, $h^{\text{SCP}}_{\text{div}}$ solves more tasks than $h^{\text{LM-cut}}$ in 32 domains, while the opposite is true in only 5 domains.

13.3 IPC 2018 planners

Diverse saturated cost partitioning heuristics also competed in the sequential optimization track of IPC 2018 as the main ingredient of the Scorpion planner (Seipp, 2018). Scorpion and $h^{\text{SCP}}_{\text{div}}$ both compute diverse saturated cost partitioning heuristics over hill climbing PDBs, systematic PDBs up to size 2 and Cartesian abstractions. The main difference between the two planners is that Scorpion uses h^2 mutexes to prune irrelevant operators (Alcázar & Torralba, 2015), an important preprocessing technique used by all top performers in IPC 2018. Another difference is that the two planners employ different time limits for diversification and optimization: $h^{\text{SCP}}_{\text{div}}$ uses 1000 seconds and 100 seconds, whereas Scorpion uses 200 seconds and 2 seconds, respectively. Also, in contrast to $h^{\text{SCP}}_{\text{div}}$, Scorpion uses strong stubborn sets to prune successor states (Alkhazraji, Wehrle, Mattmüller, & Helmert, 2012; Wehrle & Helmert, 2014).

Table 7b compares Scorpion to the three strongest non-portfolio planners from IPC 2018: Complementary 1 (Franco, Lelis, Barley, Edelkamp, Martines, & Moraru, 2018), Complementary 2 (Franco, Torralba, Lelis, & Barley, 2017) and Planning-PDBs (Moraru, Edelkamp, Martinez, & Franco, 2018). Scorpion solves more tasks than the other three planners in 29 domains, while the opposite is the case in only 12–13 domains. Scorpion also has an edge over the other three planners in terms of total coverage: it solves 121 more tasks than the strongest contender.

14. Conclusion

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.

Acknowledgments

We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639).

Appendix A. Abstraction Heuristics with General Cost Functions are Admissible and Consistent

We state in Section 5 that all abstraction heuristics are admissible and consistent even if we allow negative and infinite costs. We now show this result.

First, we recall the definition of goal distances for a transition system $\mathcal{T}$, cost function $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and state $s \in S(\mathcal{T})$:

$$h^*_{\mathcal{T}}(\mathit{cost}, s) = \inf_{\pi \in \Pi_\star(\mathcal{T}, s)} \mathit{cost}(\pi),$$

where $\Pi_\star(\mathcal{T}, s)$ is the set of goal paths from s in $\mathcal{T}$ and $\inf \emptyset$ is defined as ∞. We write h^* for $h^*_{\mathcal{T}}$ where $\mathcal{T}$ is clear from context or does not matter.

We first show that all heuristics that are goal-aware and consistent are admissible, and then we conclude the proof by showing that abstraction heuristics (with general cost functions) are goal-aware and consistent.

A.1 Goal-Aware + Consistent $\Longrightarrow$ Admissible

It is well-known that in the setting of finite non-negative cost functions, a heuristic that is both goal-aware and consistent is admissible. We now show that this result also holds for general cost functions.

Let $\mathcal{T}$ be a transition system, and let h be a heuristic for $\mathcal{T}$ that is goal-aware and consistent. We show that h is admissible.

Let $s \in S(\mathcal{T})$ and $\mathit{cost} \in \mathcal{C}(\mathcal{T})$. We must show $h(s) \leq h^*(s)$. Because $h^*(s)$ is defined as the infimum of the costs of all goal paths for s, it is sufficient to show that $h(s) \leq \mathit{cost}(\pi)$ for all goal paths π for s.

Let $\pi = \langle s \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n \rangle$ be such a goal path from s to a goal state s_n. Because h is consistent we have $h(s) \leq \mathit{cost}(\ell_1) \oplus h(s_1) \leq \mathit{cost}(\ell_1) \oplus \mathit{cost}(\ell_2) \oplus h(s_2) \leq \cdots \leq \mathit{cost}(\ell_1) \oplus \ldots \mathit{cost}(\ell_n) \oplus h(s_n) = \mathit{cost}(\pi) \oplus h(s_n)$. Since h is goal-aware, we have $h(s_n) \leq 0$ and therefore $h(s) \leq \mathit{cost}(\pi)$.

A.2 Abstraction Heuristics with General Cost Functions are Goal-Aware

Let h be an abstraction heuristic for transition system $\mathcal{T}$ with abstraction mapping α and abstract transition system $\mathcal{T}'$. Let $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and $s \in S_\star(\mathcal{T})$. We must show $h(\mathit{cost}, s) \leq 0$.

By definition of abstraction heuristics, we have $h(\mathit{cost}, s) = h^*_{\mathcal{T}'}(\mathit{cost}, \alpha(s))$. Because s is a goal state of $\mathcal{T}$, $\alpha(s)$ is a goal state of $\mathcal{T}'$. (This is one of the properties required of abstraction mappings; cf. Helmert et al., 2007.) Because the empty path is a goal path for $\alpha(s)$ with cost 0, we have $h^*_{\mathcal{T}'}(\mathit{cost}, \alpha(s)) \leq 0$, showing that h is goal-aware.

A.3 Abstraction Heuristics with General Cost Functions are Consistent

Let h be an abstraction heuristic for transition system $\mathcal{T}$ with abstract transition system $\mathcal{T}'$. Let $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and $s \xrightarrow{\ell} s' \in T(\mathcal{T})$. We must show $h(\mathit{cost}, s) \leq \mathit{cost}(\ell) \oplus h(\mathit{cost}, s')$.

Since abstractions preserve transitions (Helmert et al., 2007), we have $\alpha(s) \xrightarrow{\ell} \alpha(s') \in T(\mathcal{T}')$. Therefore, $h(\mathit{cost}, s) = h^*_{\mathcal{T}'}(\mathit{cost}, \alpha(s)) = \inf_{\pi \in \Pi_\star(\mathcal{T}', \alpha(s))} \mathit{cost}(\pi) \leq \inf_{\pi' \in \Pi_\star(\mathcal{T}', \alpha(s'))} (\mathit{cost}(\ell) \oplus \mathit{cost}(\pi')) = \mathit{cost}(\ell) \oplus \inf_{\pi' \in \Pi_\star(\mathcal{T}', \alpha(s'))} \mathit{cost}(\pi') = \mathit{cost}(\ell) \oplus h(s')$, where the inequality holds because for every goal path $\pi' \in \Pi_\star(\mathcal{T}', \alpha(s'))$ the path π that consists of ℓ followed by π' is a goal path for $\alpha(s)$. This proves the result.

Appendix B. Cost Partitioning with General Cost Functions: Consistency

Let $h_1, \ldots, h_n$ be consistent heuristics for transition system $\mathcal{T}$, let $\mathit{cost} \in \mathcal{C}(\mathcal{T})$, and let $\mathit{cost}_1, \ldots, \mathit{cost}_n \in \mathcal{C}(\mathcal{T})$ such that $\sum_{i=1}^n \mathit{cost}_i(\ell) \leq \mathit{cost}(\ell)$ for all $\ell \in \mathcal{L}(\mathcal{T})$. We must show that the cost-partitioned heuristic h for $\mathcal{T}$ is consistent under cost function cost, i.e., $h(\mathit{cost}, s) \leq \mathit{cost}(\ell) \oplus h(\mathit{cost}, s')$ for all $s \xrightarrow{\ell} s' \in T(\mathcal{T})$, where $h(\mathit{cost}, s) := \sum_{i=1}^n h_i(\mathit{cost}_i, s)$. We emphasize that we place no restrictions on the cost functions cost, $\mathit{cost}_1, \ldots, \mathit{cost}_n$, i.e., negative and (positively or negatively) infinite costs are permitted.

We obtain

$$
\begin{aligned}
h(\mathit{cost}, s) &= \sum_{i=1}^n h_i(\mathit{cost}_i, s) && \text{(definition of } h\text{)} \\
&\leq \sum_{i=1}^n (\mathit{cost}_i(\ell) \oplus h_i(\mathit{cost}_i, s')) && \text{(consistency of } h_i\text{)} \\
&\leq \sum_{i=1}^n \mathit{cost}_i(\ell) \oplus \sum_{i=1}^n h_i(\mathit{cost}_i, s') && \text{(main step of the proof, see below)} \\
&\leq \mathit{cost}(\ell) \oplus \sum_{i=1}^n h_i(\mathit{cost}_i, s') && \text{(cost partitioning condition)} \\
&= \mathit{cost}(\ell) \oplus h(s') && \text{(definition of } h\text{)},
\end{aligned}
$$

which proves the result.

It remains to show the main step of the proof: we must demonstrate LHS $\leq$ RHS, where

$$\text{LHS} = \sum_{i=1}^{n} (cost_i(\ell) \oplus h_i(cost_i, s')) \quad \text{and}$$
$$\text{RHS} = \sum_{i=1}^{n} cost_i(\ell) \oplus \sum_{i=1}^{n} h_i(cost_i, s').$$

If $cost_i(\ell)$ and $h_i(cost_i, s')$ are finite for all $1 \leq i \leq n$, then all terms in the definition of LHS and RHS are finite, in which case it is easy to see LHS = RHS. Otherwise, let $i_0 \in \{1, \ldots, n\}$ be the smallest index for which at least one of $cost_{i_0}(\ell)$ and $h_{i_0}(cost_{i_0}, s')$ is (positively or negatively) infinite.

If $cost_{i_0}(\ell) = \infty$, then $\sum_{i=1}^{n} cost_i(\ell) = \infty$ and hence RHS $= \infty$. Similarly, if $h_{i_0}(cost_{i_0}, s') = \infty$, we get $\sum_{i=1}^{n} h_i(cost_i, s') = \infty$ and again RHS $= \infty$. In both cases LHS $\leq$ RHS holds trivially because $x \leq \infty$ for all x.

So it remains to consider the case where neither $cost_{i_0}(\ell)$ nor $h_{i_0}(cost_{i_0}, s')$ equals ∞, but at least one of them is infinite. Then this must be a negative infinity, and we obtain $cost_{i_0}(\ell) \oplus h_{i_0}(cost_{i_0}, s') = -\infty$. Because i_0 is the first index for which we get infinities, we also know that $cost_j(\ell) \oplus h_j(cost_j, s')$ is finite for all $j < i_0$. Together, these two facts show LHS $= -\infty$, from which LHS $\leq$ RHS follows trivially because $-\infty \leq x$ for all x. This concludes the proof.

Appendix C. Cost Partitioning with General Cost Functions: Admissibility

Let $h_1, \ldots, h_n$ be admissible heuristics for transition system $\mathcal{T}$, let $cost \in \mathcal{C}(\mathcal{T})$, and let $cost_1, \ldots, cost_n \in \mathcal{C}(\mathcal{T})$ such that $\sum_{i=1}^{n} cost_i(\ell) \leq cost(\ell)$ for all $\ell \in \mathcal{L}(\mathcal{T})$. We must show that the cost-partitioned heuristic h for $\mathcal{T}$ is admissible under cost function $cost$, i.e., $h(cost, s) \leq h^*(cost, s)$ for all $s \in S(\mathcal{T})$, where $h(cost, s) := \sum_{i=1}^{n} h_i(cost_i, s)$. Again, no restrictions are placed on the cost functions.

We define

$$h'(cost, s) := \sum_{i=1}^{n} h^*(cost_i, s)$$

and view h' as a cost-partitioned heuristic under cost function $cost$ whose component heuristics are all h^*. The heuristic h' is clearly goal-aware, and from Appendix B, it is consistent (because h^* is consistent). As shown in Appendix A.1, h' is therefore admissible.

We get:

$$\begin{aligned}
h(cost, s) &= \sum_{i=1}^{n} h_i(cost_i, s) && \text{(definition of } h\text{)} \\
&\leq \sum_{i=1}^{n} h^*(cost_i, s) && \text{(because all } h_i \text{ are admissible)} \\
&= h'(cost, s) && \text{(definition of } h'\text{)} \\
&\leq h^*(cost, s) && \text{(because } h' \text{ is admissible),}
\end{aligned}$$

proving that h is admissible.

Appendix D. Remaining Cost Functions are Non-Negative if Original Cost Function is Non-Negative

Consider a transition system $\mathcal{T}$ and a sequence of heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ for $\mathcal{T}$. Let $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle$ be a saturated cost partitioning over $\mathcal{H}$ of a non-negative cost function $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and let $\langle \mathit{remain}_0, \ldots, \mathit{remain}_n \rangle$ be the sequence of remaining cost functions produced by the saturated cost partitioning algorithm. Then all remaining cost functions are non-negative.

Since $\mathit{remain}_0 = \mathit{cost}$ by Definition 13 and $\mathit{cost} \geq 0$, we have $\mathit{remain}_0 \geq 0$. By requirement 1 for saturated cost functions in Definition 10, we have $\mathit{cost}_i \leq \mathit{remain}_{i-1}$ and therefore $\mathit{remain}_i = \mathit{remain}_{i-1} - \mathit{cost}_i \geq 0$ for $1 \leq i \leq n$.

Appendix E. Cost-Partitioned Heuristic Values are Non-Negative if Original Cost Function is Non-Negative

Consider a transition system $\mathcal{T}$ and a sequence of abstraction heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ for $\mathcal{T}$. Let $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle$ be a saturated cost partitioning over $\mathcal{H}$ of a non-negative cost function $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ and let $\langle \mathit{remain}_0, \ldots, \mathit{remain}_n \rangle$ be the sequence of remaining cost functions produced by the saturated cost partitioning algorithm. Then $\sum_{i=1}^{n} h(\mathit{cost}_i, s) \geq 0$ for all $s \in S(\mathcal{T})$.

To see this, consider any $i \in \{1, \ldots, n\}$. By Appendix D we have $\mathit{remain}_{i-1} \geq 0$. Therefore, $h^*_{\mathcal{T}_{h_i}}(\mathit{remain}_{i-1}, s') \geq 0$ for all abstract states $s' \in S(\mathcal{T}_{h_i})$ and consequently $h_i(\mathit{remain}_{i-1}, s) \geq 0$ for all concrete states $s \in S(\mathcal{T})$. Together with requirement 2 for saturated cost functions in Definition 10, we obtain $h_i(\mathit{cost}_i, s) = h_i(\mathit{remain}_{i-1}, s) \geq 0$ for all $s \in S(\mathcal{T})$ and therefore $\sum_{i=1}^{n} h(\mathit{cost}_i, s) \geq 0$ for all $s \in S(\mathcal{T})$.

Appendix F. Minimum Saturated Cost Function for Abstraction Heuristics

Consider a weighted (concrete) transition system $\langle \mathcal{T}, \mathit{cost} \rangle$ and an abstraction heuristic h for $\mathcal{T}$ with underlying (abstract) transition system $\mathcal{T}'$. We first show that the minimum saturated cost function *mscf* for h and *cost*, as defined in Equation 1, satisfies the two properties of saturated cost functions from Definition 10 and then show that *mscf* is minimal among all saturated cost functions for h and *cost*:

Prop. 1. We must show $\mathit{mscf}(\ell) \leq \mathit{cost}(\ell)$ for all labels $\ell \in \mathcal{L}(\mathcal{T})$. We distinguish between four cases:

(1) If $\mathit{mscf}(\ell) = -\infty$ or $\mathit{cost}(\ell) = \infty$, the inequality holds trivially. (2) We have $\mathit{mscf}(\ell) = \infty$ if there is a transition $a \xrightarrow{\ell} b \in T(\mathcal{T}')$ such that one of the following conditions holds:

- $h^*_{\mathcal{T}'}(\mathit{cost}, a)$ is finite and $h^*_{\mathcal{T}'}(\mathit{cost}, b) = -\infty$
- $h^*_{\mathcal{T}'}(\mathit{cost}, a) = \infty$ and $h^*_{\mathcal{T}'}(\mathit{cost}, b)$ is finite
- $h^*_{\mathcal{T}'}(\mathit{cost}, a) = \infty$ and $h^*_{\mathcal{T}'}(\mathit{cost}, b) = -\infty$

By the definition of path addition and the triangle inequality for shortest paths in graphs, $\mathit{cost}(\ell)$ must be ∞ in all three conditions and therefore $\mathit{mscf}(\ell) = \mathit{cost}(\ell) = \infty$.

(3) For the last case involving infinities, we show by contradiction that $\mathit{cost}(\ell) = -\infty$ implies $\mathit{mscf}(\ell) = -\infty$ for all labels $\ell \in \mathcal{L}(\mathcal{T})$. Assume there is a label $\ell \in \mathcal{L}(\mathcal{T})$ with

cost$(\ell) = -\infty$ and *mscf*$(\ell) > -\infty$. Then there is a transition $a \xrightarrow{\ell} b \in T(\mathcal{T}')$, such that $h^*_{\mathcal{T}'}(\mathit{cost}, a) \neq -\infty$ and $h^*_{\mathcal{T}'}(\mathit{cost}, b) \neq \infty$. However, if $h^*_{\mathcal{T}'}(\mathit{cost}, b) \neq \infty$, we must have $h^*_{\mathcal{T}'}(\mathit{cost}, a) = -\infty$, since b is reachable from a via ℓ with cost $-\infty$, which leads to a contradiction.

Otherwise, *mscf*(ℓ) and *cost*(ℓ) are finite and there exists a transition $a \xrightarrow{\ell} b \in T(\mathcal{T}')$ such that $h^*_{\mathcal{T}'}(\mathit{cost}, a)$ and $h^*_{\mathcal{T}'}(\mathit{cost}, b)$ are finite and *mscf*$(\ell) = h^*_{\mathcal{T}'}(\mathit{cost}, a) - h^*_{\mathcal{T}'}(\mathit{cost}, b) \leq$ *cost*$(\ell) + h^*_{\mathcal{T}'}(\mathit{cost}, b) - h^*_{\mathcal{T}'}(\mathit{cost}, b) =$ *cost*(ℓ), where we use that $h^*_{\mathcal{T}'}(\mathit{cost}, a) \leq$ *cost*$(\ell) + h^*_{\mathcal{T}'}(\mathit{cost}, b)$ by the triangle inequality.

Prop. 2. We must show $h(\mathit{mscf}, s) = h(\mathit{cost}, s)$ for all states $s \in S(\mathcal{T})$.

From *mscf*$(\ell) \leq$ *cost*(ℓ) for all $\ell \in \mathcal{L}(\mathcal{T})$, we get that $h(\mathit{mscf}, s) \leq h(\mathit{cost}, s)$ for all states $s \in S(\mathcal{T})$ since lowering the weights in a transition system can only decrease goal distances.

It remains to show $h(\mathit{mscf}, s) \geq h(\mathit{cost}, s)$ for all concrete states $s \in S(\mathcal{T})$. Since changing the cost function does not affect the abstraction mapping, this is the case if $h^*_{\mathcal{T}'}(\mathit{mscf}, a) \geq h^*_{\mathcal{T}'}(\mathit{cost}, a)$ for all abstract states $a \in S(\mathcal{T}')$.

Let a_0 be any abstract state in $S(\mathcal{T}')$. If there is no goal path for a_0, we get $h^*_{\mathcal{T}'}(\mathit{mscf}, a_0) = h^*_{\mathcal{T}'}(\mathit{cost}, a_0) = \infty$.

If all goal paths for a_0 use a transition with label cost ∞, we also have $h^*_{\mathcal{T}'}(\mathit{cost}, a_0) = \infty$. We need to show that $h^*_{\mathcal{T}'}(\mathit{mscf}, a_0) = \infty$ in this case as well. Let $a \xrightarrow{\ell} b$ with *cost*$(\ell) = \infty$ be one of the transitions that is part of a shortest goal path for a_0. Since the transition is part of the shortest goal path, we know that $h^*_{\mathcal{T}'}(\mathit{cost}, a) = \infty$. If $h^*_{\mathcal{T}'}(\mathit{cost}, b) \neq \infty$, we have $h^*_{\mathcal{T}'}(\mathit{cost}, a) \ominus h^*_{\mathcal{T}'}(\mathit{cost}, b) = \infty$ and therefore *mscf*$(\ell) = \infty$ and $h^*_{\mathcal{T}'}(\mathit{mscf}, a_0) = \infty$. If $h^*_{\mathcal{T}'}(\mathit{cost}, b) = \infty$, then b is not a goal state and the shortest goal path for b uses a transition with cost ∞. Therefore, we have $h^*_{\mathcal{T}'}(\mathit{cost}, a) \ominus h^*_{\mathcal{T}'}(\mathit{cost}, b) = -\infty$ but $h^*_{\mathcal{T}'}(\mathit{mscf}, a_0)$ is still ∞.

Otherwise, we know that there is a goal path for a_0 that only uses labels ℓ with *cost*$(\ell) < \infty$. Let $\pi = \langle a_0 \xrightarrow{\ell_1} a_1, \ldots, a_{k-1} \xrightarrow{\ell_k} a_k \rangle$ be such a goal path. If *cost*$(\ell_i) = -\infty$ for a label ℓ_i on the path, we have $h^*_{\mathcal{T}'}(\mathit{cost}, a_0) = -\infty$. Since *mscf*$(\ell_i) \leq$ *cost*(ℓ_i) by the first requirement for saturated cost functions, we have *mscf*$(\ell_i) = -\infty$ and therefore $h^*_{\mathcal{T}'}(\mathit{mscf}, a_0) = -\infty$.

Otherwise, all label costs on π are finite. However, the goal distance $h^*_{\mathcal{T}'}(\mathit{cost}, a_0)$ is still $-\infty$ if there is a negative-cost cycle on π. Since reducing label costs preserves all negative-cost cycles, we have $h^*_{\mathcal{T}'}(\mathit{cost}, a_0) = h^*_{\mathcal{T}'}(\mathit{mscf}, a_0) = -\infty$ in this case.

If all label costs on π and all goal distances under *cost* are finite, we can bound the cost of π under *mscf* by

$$\begin{aligned}
\sum_{i=1}^{k} \mathit{mscf}(\ell_i) &\overset{(1)}{\geq} \sum_{i=1}^{k} (h^*_{\mathcal{T}'}(\mathit{cost}, a_{k-1}) - h^*_{\mathcal{T}'}(\mathit{cost}, a_k)) \\
&\overset{(2)}{=} \sum_{i=0}^{k-1} h^*_{\mathcal{T}'}(\mathit{cost}, a_k) - \sum_{i=1}^{k} h^*_{\mathcal{T}'}(\mathit{cost}, a_k) \\
&\overset{(3)}{=} h^*_{\mathcal{T}'}(\mathit{cost}, a_0) - h^*_{\mathcal{T}'}(\mathit{cost}, a_k) \\
&\overset{(4)}{=} h^*_{\mathcal{T}'}(\mathit{cost}, a_0) - 0 \\
&= h^*_{\mathcal{T}'}(\mathit{cost}, a_0),
\end{aligned}$$

where (1) uses that $\mathit{mscf}(\ell) \geq h^*_{\mathcal{T}'}(\mathit{cost}, a) \ominus h^*_{\mathcal{T}'}(\mathit{cost}, b)$ for all transitions $a \xrightarrow{\ell} b \in T(\mathcal{T}')$, (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 $h^*_{\mathcal{T}'}(\mathit{cost}, a_0)$, the cost of an optimal plan under *cost*. This proves $h^*_{\mathcal{T}'}(\mathit{mscf}, a) \geq h^*_{\mathcal{T}'}(\mathit{cost}, a)$ for all abstract states $a \in S(\mathcal{T}')$ with finite goal distances under *cost*, 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 $h(\mathit{cost}', s) = h(\mathit{cost}, s)$ for all concrete states $s \in S(\mathcal{T})$ and $\mathit{cost}'(\ell) < \mathit{mscf}(\ell)$ for some label $\ell \in \mathcal{L}(\mathcal{T})$. Since $\mathit{cost}'(\ell)$ can only be lower than $\mathit{mscf}(\ell)$ if $\mathit{mscf}(\ell) \neq -\infty$, this means that there exists a transition $a \xrightarrow{\ell} b \in T(\mathcal{T}')$ with $\mathit{cost}'(\ell) < h^*_{\mathcal{T}'}(\mathit{cost}, a) - h^*_{\mathcal{T}'}(\mathit{cost}, b)$. Because $h(\mathit{cost}', s) = h(\mathit{cost}, s)$ for all concrete states $s \in S(\mathcal{T})$, we also have $h^*_{\mathcal{T}'}(\mathit{cost}', c) = h^*_{\mathcal{T}'}(\mathit{cost}, c)$ for all abstract states $c \in S(\mathcal{T}')$. With this, we obtain $\mathit{cost}'(\ell) < h^*_{\mathcal{T}'}(\mathit{cost}', a) - h^*_{\mathcal{T}'}(\mathit{cost}', b)$, which violates the triangle inequality for shortest paths in graphs.

References

Alcázar, V., & Torralba, Á. (2015). A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R., Domshlak, C., Haslum, P., & Zilberstein, S. (Eds.), *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pp. 2–6. AAAI Press.

Alkhazraji, Y., Wehrle, M., Mattmüller, R., & Helmert, M. (2012). A stubborn set algorithm for optimal planning. In De Raedt, L., Bessiere, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., & Lucas, P. (Eds.), *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pp. 891–892. IOS Press.

Ball, T., Podelski, A., & Rajamani, S. K. (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.

Bellman, R. E. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*, 87–90.

Bonet, B. (2013). An admissible heuristic for SAS$^+$ planning obtained from the state equation. In Rossi, F. (Ed.), *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 2268–2274. AAAI Press.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1), 5–33.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to Algorithms*. The MIT Press.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Domshlak, C., Helmert, M., Karpas, E., & Markovitch, S. (2011). The SelMax planner: Online learning for speeding up optimal planning. In *IPC 2011 planner abstracts*, pp. 108–112.

Doran, J. E., & Michie, D. (1966). Experiments with the graph traverser program. *Proceedings of the Royal Society A*, *294*, 235–259.

Edelkamp, S. (2001). Planning with pattern databases. In Cesta, A., & Borrajo, D. (Eds.), *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, pp. 84–90. AAAI Press.

Edelkamp, S. (2006). Automated creation of pattern database search heuristics. In Edelkamp, S., & Lomuscio, A. (Eds.), *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, pp. 35–50.

Franco, S., Lelis, L. H. S., Barley, M., Edelkamp, S., Martines, M., & Moraru, I. (2018). The Complementary1 planner in the IPC 2018. In *Ninth International Planning Competition (IPC-9): planner abstracts*, pp. 28–31.

Franco, S., Torralba, Á., Lelis, L. H. S., & Barley, M. (2017). On creating complementary pattern databases. In Sierra, C. (Ed.), *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pp. 4302–4309. IJCAI.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann.

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

Haslum, P., Botea, A., Helmert, M., Bonet, B., & Koenig, S. (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)*, pp. 1007–1012. AAAI Press.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Helmert, M., & Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway?. In Gerevini, A., Howe, A., Cesta, A., & Refanidis, I. (Eds.), *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pp. 162–169. AAAI Press.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In Boddy, M., Fox, M., & Thiébaux, S. (Eds.), *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pp. 176–183. AAAI Press.

Helmert, M., Haslum, P., Hoffmann, J., & Nissim, R. (2014). Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM*, *61*(3), 16:1–63.

Holte, R. C., Felner, A., Newton, J., Meshulam, R., & Furcy, D. (2006). Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, *170*(16–17), 1123–1136.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E., & Thatcher, J. W. (Eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum Press.

Karpas, E., & Domshlak, C. (2009). Cost-optimal planning with landmarks. In Boutilier, C. (Ed.), *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pp. 1728–1733. AAAI Press.

Karpas, E., Katz, M., & Markovitch, S. (2011). When optimal is just not good enough: Learning fast informative action cost partitionings. In Bacchus, F., Domshlak, C., Edelkamp, S., & Helmert, M. (Eds.), *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, pp. 122–129. AAAI Press.

Katz, M., & Domshlak, C. (2007). Structural patterns of tractable sequentially-optimal planning. In Boddy, M., Fox, M., & Thiébaux, S. (Eds.), *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pp. 200–207. AAAI Press.

Katz, M., & Domshlak, C. (2008). Optimal additive composition of abstraction-based admissible heuristics. In Rintanen, J., Nebel, B., Beck, J. C., & Hansen, E. (Eds.), *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pp. 174–181. AAAI Press.

Katz, M., & Domshlak, C. (2010). Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, *174*(12–13), 767–798.

Korte, B., & Vygen, J. (2001). *Combinatorial Optimization: Theory and Algorithms* (2nd edition). Springer.

Lelis, L. H. S., Franco, S., Abisrror, M., Barley, M., Zilles, S., & Holte, R. C. (2016). Heuristic subset selection in classical planning. In Kambhampati, S. (Ed.), *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pp. 3185–3191. AAAI Press.

Moraru, I., Edelkamp, S., Martinez, M., & Franco, S. (2018). Planning-PDBs planner. In *Ninth International Planning Competition (IPC-9): planner abstracts*, pp. 69–73.

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

Pisinger, D., & Ropke, S. (2010). Large neighborhood search. In Gendreau, M., & Potvin, J.-Y. (Eds.), *Handbook of Metaheuristics*, pp. 399–419. Springer.

Pommerening, F., Helmert, M., Röger, G., & Seipp, J. (2015). From non-negative to general operator cost partitioning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pp. 3335–3341. AAAI Press.

Pommerening, F., Röger, G., & Helmert, M. (2013). Getting the most out of pattern databases for classical planning. In Rossi, F. (Ed.), *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 2357–2364. AAAI Press.

Pommerening, F., Röger, G., Helmert, M., & Bonet, B. (2014). LP-based heuristics for cost-optimal planning. In Chien, S., Fern, A., Ruml, W., & Do, M. (Eds.), *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pp. 226–234. AAAI Press.

Pommerening, F., Röger, G., Helmert, M., Cambazard, H., Rousseau, L.-M., & Salvagnin, D. (2019). Lagrangian decomposition for optimal cost partitioning. In Lipovetzky, N., Onaindia, E., & Smith, D. E. (Eds.), *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, pp. 338–347. AAAI Press.

Russell, S., & Norvig, P. (1995). *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Seipp, J. (2017). Better orders for saturated cost partitioning in optimal classical planning. In Fukunaga, A., & Kishimoto, A. (Eds.), *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, pp. 149–153. AAAI Press.

Seipp, J. (2018). Fast Downward Scorpion. In *Ninth International Planning Competition (IPC-9): planner abstracts*, pp. 77–79.

Seipp, J., & Helmert, M. (2013). Counterexample-guided Cartesian abstraction refinement. In Borrajo, D., Kambhampati, S., Oddi, A., & Fratini, S. (Eds.), *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, pp. 347–351. AAAI Press.

Seipp, J., & Helmert, M. (2014). Diverse and additive Cartesian abstraction heuristics. In Chien, S., Fern, A., Ruml, W., & Do, M. (Eds.), *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pp. 289–297. AAAI Press.

Seipp, J., & Helmert, M. (2018). Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, *62*, 535–577.

Seipp, J., & Helmert, M. (2019). Subset-saturated cost partitioning for optimal classical planning. In Lipovetzky, N., Onaindia, E., & Smith, D. E. (Eds.), *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, pp. 391–400. AAAI Press.

Seipp, J., Keller, T., & Helmert, M. (2017). A comparison of cost partitioning algorithms for optimal classical planning. In Barbulescu, L., Frank, J., Mausam, & Smith, S. F. (Eds.), *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pp. 259–268. AAAI Press.

Seipp, J., Pommerening, F., & Helmert, M. (2015). New optimization functions for potential heuristics. In Brafman, R., Domshlak, C., Haslum, P., & Zilberstein, S. (Eds.), *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pp. 193–201. AAAI Press.

Seipp, J., Pommerening, F., Sievers, S., & Helmert, M. (2017). Downward Lab. `https://doi.org/10.5281/zenodo.790461`.

Sievers, S., Wehrle, M., & Helmert, M. (2014). Generalized label reduction for merge-and-shrink heuristics. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, pp. 2358–2366. AAAI Press.

Sievers, S., Wehrle, M., & Helmert, M. (2016). An analysis of merge strategies for merge-and-shrink heuristics. In Coles, A., Coles, A., Edelkamp, S., Magazzeni, D., & Sanner, S. (Eds.), *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, pp. 294–298. AAAI Press.

Wehrle, M., & Helmert, M. (2014). Efficient stubborn sets: Generalized algorithms and selection strategies. In Chien, S., Fern, A., Ruml, W., & Do, M. (Eds.), *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pp. 323–331. AAAI Press.

Yang, F., Culberson, J., Holte, R., Zahavi, U., & Felner, A. (2008). A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, *32*, 631–662.