Online Refinement of Cartesian Abstraction Heuristics

Rebecca Eifler and **Maximilian Fickert**
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
{eifler,fickert}@cs.uni-saarland.de

Abstract

In classical planning as heuristic search, the guiding heuristic function is typically treated as a black box. While many heuristics support *refinement* operations, they are typically only used for its initialization before search, but further refinement during search could make use of additional information not available in the initial state. We explore online refinement for additive Cartesian abstraction heuristics. These abstractions are computed through counter-example guided abstraction refinement, which can be applied online as well to further improve the abstractions. We introduce three operations, *refinement*, *merging*, and *reordering*, which are combined to a converging online-refinement algorithm. We describe how online refinement can effectively be used in A* and evaluate our approach on the IPC benchmarks, where it outperforms offline-generated abstractions in many domains.

Introduction

Heuristic search is one of the most successful approaches to classical planning. Many heuristics have a parameter to increase the level of precision which typically implies a trade-off with respect to the computational complexity when evaluating the heuristic. For *abstraction heuristics* (Edelkamp 2001; Helmert et al. 2014; Seipp and Helmert 2013), the size of the abstraction can be chosen to range from the null heuristic $h^0 = 0$ to the perfect heuristic h^*. *Partial delete relaxation heuristics* (Keyder, Hoffmann, and Haslum 2014; Domshlak, Hoffmann, and Katz 2015; Fickert, Hoffmann, and Steinmetz 2016) interpolate between fully relaxed semantics and real semantics.

These heuristics are often instantiated through iterative *refinement* operations. The heuristic starts out with a basic relaxation, and is repeatedly refined until a time or memory bound is reached. Given sufficiently large bounds the heuristic may *converge*, making the relaxation exact (e.g. (Haslum et al. 2007; Seipp and Helmert 2013; Helmert et al. 2014; Keyder, Hoffmann, and Haslum 2014). This process is traditionally done offline, i.e. once before search, and the resulting heuristic is treated as a black box throughout search.

However, as search progresses and new information becomes available, this additional knowledge might be used to further improve the heuristic, e.g. to eliminate flaws in the relaxation that were not apparent in the initial construction of the heuristic and were only detected later in the search process. Additional refinement steps performed online can address such issues and further improve the heuristic.

So far, online refinement of heuristic functions is mostly unexplored. Fickert and Hoffmann (2017) introduced online refinement for the $h^{C\text{FF}}$ heuristic in an enforced hill-climbing setting. The heuristic is refined whenever search is stuck in a local minimum, thus effectively removing local minima from the search space surface instead of attempting to escape them through brute-force search.

There are several other forms of online learning that do not refine a heuristic function. One such technique is updating values on a per-state basis, e.g. in transposition tables (Akagi, Kishimoto, and Fukunaga 2010) or LRTA* (Korf 1990). Similarly, Wilt and Ruml (2013) use backward search to improve the heuristic estimation: Since the h^* value is known for a backward expanded node, it can be used to compute the minimal error of the heuristic and use it to update the heuristic values during forward search. Another example is refining combinations of multiple heuristics (e.g. (Felner, Korf, and Hanan 2004; Katz and Domshlak 2010)), but the individual heuristics remain unchanged.

In this work we introduce online refinement of additive Cartesian abstraction heuristics (Seipp and Helmert 2014). The refinement operation for these heuristics is based on splits of abstract states, which allows a locally restricted refinement in small steps and is well suited for online refinement. Seipp briefly touched online refinement in his Master's Thesis (Seipp 2012), but the explored design space is small and the approach was restricted to single abstractions.

Our online-refinement algorithm defines three basic operations: *refinement*, *merging* and *reordering*. Refinement extends individual abstractions, using the same procedure that is also applied in offline refinement. The merge operation is necessary to preserve convergence against h^* when multiple additive abstractions are used. Finally, the reordering operation provides an alternative way to improve the heuristic by generating new orderings for the cost partitioning, as different orders are useful in different states (Seipp, Keller, and Helmert 2017). We combine these three operations to a monotone online-refinement procedure that converges to h^*.

We show how online refinement of Cartesian abstraction heuristics can be used in A* (Hart, Nilsson, and Raphael

1968) to improve the heuristic during search. We evaluate our approach on the IPC benchmarks and compare it to offline-generated Cartesian abstraction heuristics.

Preliminaries

In the following we consider classical planning using the finite-domain representation (FDR) (Bäckström 1995). A planning task is a 5-tuple $\Pi = (V, A, c, I, G)$, where

- V is a finite set of *state variables* where each $v \in V$ has a finite domain $\mathcal{D}(v)$. A variable/value pair $v = d$ with $v \in V$ and $d \in \mathcal{D}(v)$ is called a *fact*.
- A is a finite set of *actions*. Each action $a \in A$ is a pair $(\mathsf{pre}_a, \mathsf{eff}_a)$ of partial variable assignments which are called preconditions and effects respectively.
- $c : A \mapsto \mathbb{R}_0^+$ is the *cost function*, mapping each action to a non-negative real number.
- I is a complete assignment of variables describing the *initial state*.
- G is a partial assignment of variables describing the *goal*.

The *state space* of Π is the labeled transition system $\Theta_\Pi = (S, L, c, T, I, S^G)$. The *states* S are the complete variable assignments. The value of a variable in a state $s \in S$ is denoted by $s(v)$. An action is *applicable* in a state s if $\mathsf{pre}_a \subseteq s$. In this case, the values for all variables $v \in V$ in the state $appl(s, a)$ resulting from applying a in s are defined as $appl(s, a)(v) := \mathsf{eff}_a(v)$ if $\mathsf{eff}_a(v)$ is defined and $appl(s, a)(v) := s(v)$ otherwise. The labels L of the state space correspond to the actions A and the cost function c to that of Π. The transition relation $T \subseteq S \times L \times S$ is defined as $T = \{s \xrightarrow{a} appl(s, a) \mid \mathsf{pre}_a \subseteq s\}$. The initial state I is the same as in Π. The *goal states* $S^G = \{s \in S \mid G \subseteq s\}$ are the states that satisfy G. A *plan* for Π is an iteratively applicable sequence of actions which starts in I and leads to a goal state $s \in S^G$. A plan is *optimal* if the summed up cost of all actions is minimal among all plans of I.

A heuristic function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ maps each state to a non-negative real number or ∞. We write $h[c_i]$ to denote that the heuristic h is computed on a modification of Π where the cost function c is replaced by c_i. The *perfect heuristic* h^* assigns each state s its *remaining cost*, which is the cost of an optimal plan for s, or ∞ if no plan for s exists. A heuristic h is *admissible* if $h(s) \leq h^*(s)$ for all $s \in S$ and *consistent* if $h(s) \leq h(s') + c(a)$ for all transitions $s \xrightarrow{a} s'$. Given the transition system $\Theta = (S, L, c, T, I, S^G)$, an *abstraction* of Θ is a surjective function $\alpha : S \mapsto S^\alpha$. The *abstract state space* induced by α, written Θ^α, is the transition system $\Theta^\alpha = (S^\alpha, L, c, T^\alpha, I^\alpha, S^{\alpha G})$ with $I^\alpha = \alpha(I)$, $S^{\alpha G} = \{\alpha(s) \mid s \in S^G\}$ and $T^\alpha = \{(\alpha(s), l, \alpha(t)) \mid (s, l, t) \in T\}$. By $\sim^\alpha$ we denote the *induced equivalence relation* on Θ, defined by $s \sim^\alpha t$ iff $\alpha(s) = \alpha(t)$ and the equivalence classes by $[s]$. The *heuristic function induced by* α, written h^α, is the heuristic function which maps each state $s \in S$ to $h^*_{\Theta^\alpha}(\alpha(s))$.

A *cost partitioning* for a planning task with actions A is a set of functions $\mathcal{C} = \{c_1, \ldots, c_n : A \mapsto \mathbb{R}_0^+\}$ such that for all $a \in A : \sum_{i=1}^n c_i(a) \leq c(a)$. We say that an admissible heuristic h has a *local error* in state $s \in S$ if the *Bellman optimality equation* $h(s) \geq min_{(s,a,s') \in T} h(s') + c(a)$ is not satisfied for s. In this case we know that $h(s) < h^*(s)$, so the heuristic value could be increased in s without losing admissibility.

Additive Cartesian Abstraction Heuristics

An abstraction is *Cartesian* if all its states are Cartesian sets, i.e., they have the form $A_1 \times \cdots \times A_n$, where $A_i \subseteq \mathcal{D}(v_i)$ for all $1 \leq i \leq n$. The abstraction is built starting with the trivial abstraction and iteratively splitting states using counterexample-guided abstraction refinement (Seipp and Helmert 2013), which we summarize in the following.

In every iteration, an optimal solution as a trace $\tau = \langle [s'_0], a_1, \ldots, [s'_{n-1}], a_n, [s'_n] \rangle$, an alternating sequence of abstract states and actions, is computed. If no solution exists, the problem is unsolvable. Otherwise we check if τ can be converted to a solution of the concrete state space. During iteratively applying the actions in τ, resulting in a sequence of concrete states $s_0, s_1, \ldots, s_n$, we check if one of the following flaws occurs:

1. The concrete state s_i does not fit the abstract state $[s'_i]$ in τ, i.e. $[s_i] \neq [s'_i]$.
2. The concrete trace is completed, but s_n is not a goal state.
3. The action a_{i+1} is not applicable in the concrete state s_i.

If none of the flaws occurs, we found a solution. Otherwise, a state can be split according to the following rules (the numbers correspond to the cases above):

1. Split $[s_{i-1}]$ into $[t']$ and $[u']$ such that $s_{i-1} \in [t']$ and a_i does not lead from a state in $[t']$ to a state in $[s'_i]$.
2. Split $[s_n]$ into $[t']$ and $[u']$ such that $s_n \in [t']$ and $[t']$ does not contain a goal state.
3. Split $[s_i]$ into $[t']$ and $[u']$ in such a way that $s_i \in [t']$ and a_{i+1} is inapplicable in all states in $[t']$.

As the size of the abstract state space grows larger, the number of refinement iterations that are necessary to result in an increase of the heuristic estimate also becomes larger. In order to avoid this problem, a set of multiple small abstractions can be used instead. Multiple abstractions can be generated by only considering one goal fact in each abstraction, such that each abstraction covers different parts of the planning task (Seipp and Helmert 2014).[1]

Cost partitionings can be used to admissibly combine a set of heuristics. The *saturated cost partitioning (SCP)* is an effective way to construct an additive ensemble of multiple Cartesian abstractions (Seipp and Helmert 2014).

For a heuristic h and cost function c, the *saturated cost function saturate*(h, c) is defined as the minimal cost function $c' \leq c$ with $h[c'](s) = h[c](s)$ for all states s. Given a set of heuristic functions $\mathcal{H} = \{h_1, \cdots, h_n\}$ for Π and an order $\omega = (h_1, \cdots, h_n)$ of those functions, the *saturated*

[1] Seipp and Helmert also define a decomposition based on landmarks, which we do not consider here as it requires non-trivial extensions to the online refinement and merging procedures.

cost partitioning $\mathcal{C} = c_1, \cdots c_n$ and the *remaining cost functions* $\bar{c}_0, \cdots, \bar{c}_n$ are defined as

$$\bar{c}_0 = c$$
$$c_i = \mathit{saturate}(h_i, \bar{c}_{i-1})$$
$$\bar{c}_i = \bar{c}_{i-1} - c_i$$

If h is an abstraction heuristic based on an abstract transition system T^α of Π with labels L, then the saturated cost function $\hat{c}(a)$ for $a \in L$ is defined as $\hat{c}(a) = \max_{s \xrightarrow{a} s' \in T^\alpha} \max\{0, h(s) - h(s')\}$. This ensures that each abstraction only uses the minimal amount of cost required to preserve the cost of an optimal plan from each state.

Running Example Our example consists of a robot who has to visit certain cells on a small grid (Figure 1).

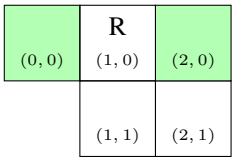

Figure 1: Sample task: the robot R must visit the green cells.

The state variables are the robot position *at* (which can be any of the five locations, initially $(1, 0)$) and the boolean variables v_{00} and v_{20} indicating if the corresponding cells have been visited (initially 0, must be 1 in the goal). The robot can move between adjacent cells $x, y \in \mathcal{D}(at), x \neq y$ with a move action $m(x, y)$ with preconditions $\{at = x\}$ and effects $\{at = y\}$. If the target position of the move is either one of the goal locations ($y = (0, 0)$ or $y = (2, 0)$), the effects include achieving the corresponding visited fact ($v_{00} = 1$ or $v_{20} = 1$ respectively). All action costs are 1, except the move action from $(1, 0)$ to $(1, 1)$, which costs 2.

The offline-refined Cartesian abstractions of the example are shown in Figure 2. The procedure starts with a trivial abstraction of a single abstract state for each goal. Initially, the abstract solution is empty because the abstract initial state is already a goal state. To prevent this flaw, the abstract state is split on the goal fact $v_{00} = 1$ respectively $v_{20} = 1$. This results in the abstractions shown in Figure 2. Since now the abstract solution corresponds to the concrete solution in the individual goal abstractions, the refinement terminates.

(a) $\mathcal{A}_1$ (b) $\mathcal{A}_2$

Figure 2: Abstractions of the running example after offline refinement. If a variable is not mentioned in a state all values are possible. Self loops are omitted. The SCP order is $\omega = \{\mathcal{A}_1, \mathcal{A}_2\}$. Goal states are marked in green. Actions are abbreviated, e.g. as $m(10, 00)$ instead of $m((1, 0), (0, 0))$.

Online Refinement Operations

In the following, we describe the three operations *refinement*, *merging*, and *reordering*, that make up our online-refinement approach.

Refinement of Additive Cartesian Abstractions

The *refinement* operation is based on the refinement algorithm described in the previous section. The essential modification for online refinement is the start state of the trace τ. While offline refinement always starts from the initial state, online refinement uses the current search state. If the solution for each individual goal is short, but the goals influence each other strongly, the abstractions refined offline largely underestimate the remaining cost. The reason is that an abstraction refined offline does not consider going into a wrong direction first, and states that are not on an optimal path for the initial state in the abstraction are never refined further.

If the sample abstractions are refined on the state $s_{ru} = \{at = (2, 0), v_{00} = 0, v_{20} = 1\}$ where the robot is in the right upper cell, $\mathcal{A}_1$ changes as shown in Figure 3. The action $m(10, 00)$ of the abstract solution is not applicable in s_{ru}, so the starting cell of the robot is split from the other cells. As a result, the heuristic value of the refined state increases from 1 to 2. The abstraction $\mathcal{A}_2$ does not change because it is refined on a goal state.

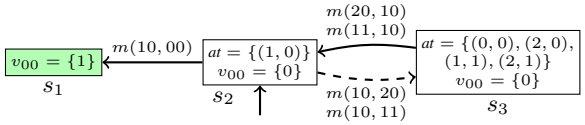

Figure 3: $\mathcal{A}_1$ after online refinement on the state $\{at = (2, 0), v_{00} = 1, v_{20} = 0\}$. Solid transitions have cost 1, dashed ones have cost 0.

Influence on Cost Partitioning After every refinement of the abstractions the cost partitioning needs to be recomputed. Here, two undesirable effects can occur. The first abstraction absorbs more and more of the cost. Thereby the impact of the additive component of the abstractions at the end of the cost partitioning order is diminished. Secondly, it is possible that the heuristic estimation of a state decreases after the cost is redistributed by the saturated cost partitioning algorithm, as illustrated in the following example.

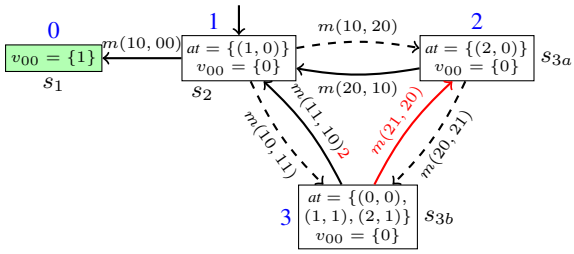

Figure 4: $\mathcal{A}_1$ after online refinement on state $at = (2, 1), v_{00} = 0, v_{20} = 1$. The blue numbers correspond to the remaining cost of the state.

If $\mathcal{A}_1$ in Figure 3 is further refined on the state $s_{rl} = \{at = (2, 1), v_{00} = 0, v_{20} = 1\}$, state s_3 is split as shown in Figure 4. The first action of the abstract solution $m(20, 10)$ is not applicable in s_{rl}, so the precondition $at = (2, 0)$ is split from s_3. The solid arrows indicate the actions which

retain their cost after the *saturation* of the abstraction. As the action $m(10, 11)$ has a cost of 2, the cost of the action $m(21, 20)$ is necessary to preserve the optimal plan cost of s_{3b}. As a result, there is no cost for $m(10, 20)$ remaining in $\mathcal{A}_2$, which now evaluates to 0 for any state. Overall, in the additive heuristic of $\mathcal{A}_1$ and $\mathcal{A}_2$, the heuristic estimation for the states abstracted by s_{3b} in Figure 4 increased by 1, while for all others it decreased by 1.

Both problems, the dominance of the first abstractions in the order and the decreasing estimation, can be solved by a slight adaptation of the SCP algorithm. Instead of completely redistributing the cost, every abstraction can keep the cost of the previous iteration, and only gains new cost from the cost which is not used by any other abstraction in the previous iteration. In the following, this cost is called *unused cost*. For the abstraction in Figure 4 this means that it can not use the cost of the action $m(21, 20)$, because it is already used by $\mathcal{A}_2$. Therefore, the heuristic estimation does not decrease for any state.

Definition 1 *Given the cost partitioning* $\mathcal{C}^{l-1} = \{c_1^{l-1}, \cdots, c_n^{l-1}\}$ *of the previous iteration, the* online saturated cost partitioning (OSCP) $\mathcal{C}^l = \{c_1^l, \ldots, c_n^l\}$ *and the remaining cost functions* $\bar{c}_0^l, \ldots, \bar{c}_n^l$ *are defined as*

$$\bar{c}_0^l = c - \sum_{j=1}^{n} c_j^{l-1} \quad \textit{(unused cost)}$$

$$c_i^l = \textit{saturate}(h_i, \bar{c}_{i-1}^l + c_i^{l-1})$$

$$\bar{c}_i^l = \bar{c}_{i-1}^l - c_i^l$$

Useful Splits Every split of an abstract state increases the memory size of the abstraction and the evaluation time of the heuristic. Hence, it is only useful to split an abstract state if this could increase the heuristic value of some state. If a state s is split into the states s' and s'', the heuristic can only increase if the cost of all actions in at least one direction between s' and s'' is greater than 0. Otherwise, it is still possible to move between these states for free and the split has no impact on the remaining cost of any abstract state. Exactly this behavior happens in the split of state s_3 in Figures 3 and 4. When performing the OSCP, none of the actions between the states s_{3a} and s_{3b} has a cost larger than zero. Therefore, the heuristic estimation can not increase.

In the following, a split of a state s is called *useful*, if all actions in at least one direction between the resulting states s' and s'' have a non-zero cost after recomputing the cost partitioning. The check if a split is useful is implemented by testing if there is still *unused cost* or cost reserved by the abstraction (in any order in $\mathcal{O}$, c.f. Section Reordering) for all actions in at least one direction between s' and s''. Since a non-useful split can sometimes be necessary to make a useful split reachable in refinement, it is possible that the useful split check prevents heuristic from increasing.

Merging

Originally the reason to use multiple small abstractions instead of one large abstraction was a slow increase in the heuristic estimation. But this separation of the goal facts prevents a convergence of the heuristic against h^*. This behavior can be observed for the initial state of the sample task. The heuristic value based on the two abstractions will never be 3 for the initial state, independent of the number of refinement operations. We can restore this convergence property by replacing two abstractions $\mathcal{A}_1$ and $\mathcal{A}_2$ by their *synchronized product* whenever further improvement based on refinement it not possible. The synchronized product are the non-empty intersections of the abstract states of $\mathcal{A}_1$ and $\mathcal{A}_2$. The merge result is Cartesian because the intersection of two Cartesian sets is again Cartesian (Seipp 2012).

Considering again our example, the synchronized product of $\mathcal{A}_1$ (Figure 3) and $\mathcal{A}_2$ (Figure 2) is shown in Figure 5.

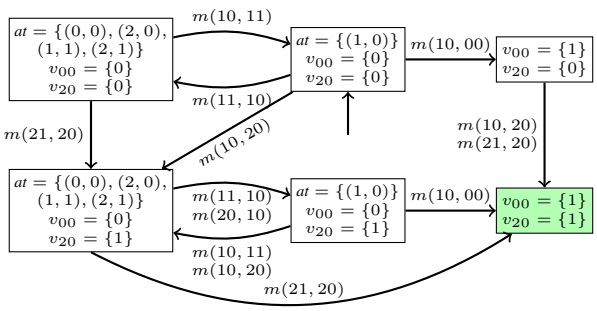

Figure 5: *Synchronized product* of the abstractions $\mathcal{A}_1$ (Figure 3) and $\mathcal{A}_2$ (Figure 2).

While the merge operation itself does not change the heuristic value, it allows further refinement operations to be performed on the resulting abstraction. Afterwards, the cost partitioning for all abstractions is recomputed using the sum of the cost functions of $\mathcal{A}_1$ and $\mathcal{A}_2$ as the cost function of the merge result.

Reordering

The order in which the cost functions for the saturated cost partitioning are computed can have a huge impact on the informativeness of the heuristic. The performance of the heuristic can be improved by using a set of orders $\mathcal{O}$. When evaluating a state, the heuristic can use the maximum estimation of all cost partitionings corresponding to the orders $\mathcal{O}$ (Seipp, Keller, and Helmert 2017). Diverse orders are obtained by generating several potentially useful orders, and only retaining those that lead to an improved estimation on at least one randomly sampled state.

These approaches can be transferred to the online phase to potentially gain better orders, because instead of random sample states, actual search states can be used.

We start out with one order based on the h^{add} value of the goal fact of the abstractions, following the default configuration by Seipp and Helmert (2014). If, during search, an order leading to a higher estimation for the current search state is found, it is added to $\mathcal{O}$ and can be used in all following states. If the structure of any abstraction changes, either by refinement or merging, the cost partitioning for each order $\omega \in \mathcal{O}$ is recomputed through OSCP.

When generating a new order, we order the abstractions by their impact on the current search state. More specifically, the abstractions are ordered descendingly according to their individual goal distance, using the original cost function. This strategy worked best in preliminary experiments.

Converging Online Refinement

We now describe our converging online-refinement procedure that combines the three introduced operations (Algorithm 1). Our approach relies on the Bellman equation to identify states with a local error, which the online refinement algorithm aims to correct.

Algorithm 1: Online Refinement

Input: An additive Cartesian abstraction heuristic h with abstractions $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and orders $\mathcal{O}$, and a state s where h does not satisfy Bellman

$\omega' :=$ FINDORDER(h, s)
$c_{\omega'} :=$ SCP(h, ω')
if $h(s)$ *increases when using* $c_{\omega'}$ **then**
  $\mathcal{O} := \mathcal{O} \cup \{\omega'\}$
while $\neg$BELLMAN(h, s) **do**
  for $i := 1, \ldots, n$ **do**
    REFINE$(\mathcal{A}_i, s)$
  if *no abstraction* $\mathcal{A}_i$ *was modified* **then**
    Let $\mathcal{A}_x, \mathcal{A}_y$ be the two abstractions in h with the fewest abstract states
    MERGE$(\mathcal{A}_x, \mathcal{A}_y)$
  for $\omega \in \mathcal{O}$ **do**
    OSCP(h, ω)

First the algorithm tries to improve the heuristic by finding a better cost partitioning order for the current state. If this does not suffice to satisfy the Bellman equation, all abstractions are refined on the current search state until either there is no local error anymore or no further refinement is possible. In the latter case, the two smallest abstractions are merged to gain new refinement opportunities.

In preliminary experiments, we found that the refine operation has a higher impact on performance than the reorder operation when only using these operations individually. The combination of both works best, while the order in which the operations are executed is not important. The merge operation does not significantly improve the heuristic by itself even though it is the operation which introduces the most overhead. However, it is needed to guarantee convergence.

Theoretical Properties

Theorem 1 *Let $\Pi = (V, A, c, I, G)$ be a planning task, $\mathcal{H}$ be a set of Cartesian Abstraction heuristics, and $\mathcal{O}$ a set of orderings for $\mathcal{H}$. Then the heuristic estimation for any state can not decrease after applying any of the introduced operations (refining (i), merging (ii) or reordering (iii)) and subsequent recomputation of the cost partitioning for (i) and (ii).*

Proof Sketch:
For (i): Refinement of an abstraction without changing the cost function is monotone. As the OSCP does not decrease the cost of any action (unless decreasing the cost preserves the optimal plan cost for all states) it can not lead to a cheaper solution for any state. If the refinement of all heuristics is monotone then so is the sum of them.
For (ii): For any state s, an optimal plan p for s in the synchronized product of two abstractions $\mathcal{A}_1$ and $\mathcal{A}_2$ is also a (not necessarily optimal) plan in both $\mathcal{A}_1$ and $\mathcal{A}_2$. Let c_i be the cost function of $\mathcal{A}_i$ and c_M of the merge result. Then it suffices to show that $\sum_{a \in p} c_M(a) \geq \sum_{a \in p} c_1(a) + \sum_{a \in p} c_2(a)$ because an optimal plan in $\mathcal{A}_i$ is as most as expensive as p. The inequality holds since c_M is defined as $c_M = c_1 + c_2$. The recomputation of the cost partitioning is monotone as shown in (i).
For (iii): As orders are only added to $\mathcal{O}$ and we always take the maximum estimation of all orders, the estimation can only increase for any state.

Theorem 2 *Let $\Pi = (V, A, c, I, G)$ be a planning task, and h an additive Cartesian abstraction heuristic. Then using the refinement procedure described in Algorithm 1 the heuristic converges towards h^*.*

Proof Sketch:
Whenever no further refinement operations are possible, two abstractions are merged. If necessary, in the end this results in one big abstraction containing all goal facts. This leads to a convergence against h^* in every planning task, as the optimal plan in the merged abstraction will also be an optimal plan in the original task in the limit.

Online Refinement in A^*

The A* search algorithm needs one adaption to handle a dynamically changing heuristic. The open list stores the search nodes according to the sum of the heuristic estimation and the shortest known distance from the initial state. When the heuristic function changes, the open list must be resorted in order to always use the best known estimation in the expansion order. Other possibilities would be to restart the search or spawning parallel search processes (Arfaee, Zilles, and Holte 2011). Both approaches seem unsuitable if the heuristic changes frequently but locally restricted. Not updating the open list would lead to an admissible but *inconsistent* heuristic resulting in reopened search nodes. As the heuristic function can only increase for any state (Theorem 1), it is not necessary to reorder the entire open list. Instead, we can do this lazily: Every time a state is expanded, we check if the heuristic value using the current heuristic is the same as the one when the state was inserted into the open list. If this is the case the state is expanded, otherwise it is reinserted into the open list with the updated heuristic value. Whenever a state that is currently being expanded has a local error, the refinement procedure is called because we know that the heuristic value can be increased in that state.

Experiments

We implemented our techniques in Fast Downward (FD) (Helmert 2006) based on the existing implementation of Cartesian abstraction heuristics (Seipp and Helmert 2013;

2014). The experiments were run on Intel Xenon E5-2650 v3 processors with a clock rate of 2.3 GHz. The time and memory limit were set to 30 minutes and 4 GB. As benchmarks we use all domains from the optimal tracks of all IPCs up to 2014 (excluding the trivial Movie domain), for a total of 1637 problem instances.

First, we look at the search behavior of our online-refinement algorithm and analyze the overhead added by online-refinement. As this overhead can sometimes be prohibitive, we devise additional configurations in an attempt to reduce this. We compare our configurations to offline-refined Cartesian abstraction heuristics.

Overview

For our base configuration h^{on}, we initialize the heuristic with 1000 (offline-refined) abstract states in total. During search, we apply our online-refinement procedure in each state until the Bellman equation is satisfied.

As our comparison baseline, we use an offline-refined heuristic h^{off} with a refinement timeout of 15 minutes. The cost partitioning order uses the default setting of a descending order of the h^{add} values of the goal facts that correspond to the individual abstractions.

In the two leftmost columns of Table 1, the coverage of both configurations is displayed. The online-refinement version solves a total of 532 tasks, 204 less than the offline version. In 32 domains h^{off} solves more tasks than h^{on}, in 3 domains they solve equally many, and in 4 domains h^{on} solves more tasks. Our online refinement approach works best in the Miconic domain, where it solves 104 instances compared to 63 with h^{off}. Online refinement seems unsuitable for the Openstacks, Pegsol, and Tidybot domains, as the overall coverage drops by 31, 40, and 23 respectively.

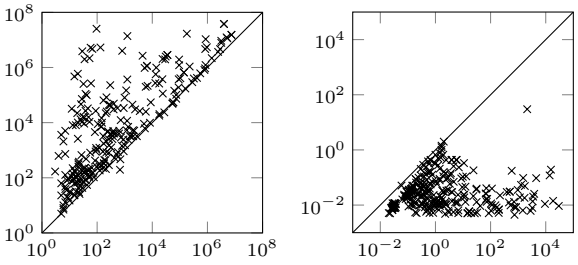

Figure 6: Left: Number of expansions until the last f-layer. Right: Search time per expanded state per task in ms. The x- and y-axes correspond to h^{on} respectively h^{off}.

The left side of Figure 6 compares the number of expansions until the last f-layer is reached for commonly solved instances of h^{on} and h^{off}. With very few exceptions, h^{on} needs significantly fewer expansions, up to 5 orders of magnitude fewer in some instances. On larger instances, this observation becomes more pronounced, as the heuristic is more frequently refined and in the end much more informative than h^{off}. Since initially h^{off} may have more abstract states than h^{on}, on some (very few) smaller instances there are cases where more expansions are necessary.

Domain	h^{on}	h^{off}	$h^{\text{on}}_{0.1}$	*exp*	*time*	h^{SCP}_{div}
Airport (50)	23	**34**	33	0.02	0.57	30
Barman (34)	0	**4**	**4**	1.02	2.90	**4**
Blocksworld (35)	10	18	22	0.24	1.07	**28**
Childsnack (20)	0	0	0	–	–	0
Depot (22)	2	5	9	0.11	0.38	**11**
Driverlog (20)	8	11	**14**	0.08	0.41	**14**
Elevators (50)	36	37	42	0.52	1.43	**44**
Floortile (40)	0	2	**4**	0.22	1.08	2
FreeCell (80)	6	19	20	0.11	1.04	**65**
GED (20)	5	15	**16**	1.09	4.37	15
Grid (5)	1	2	**3**	0.02	0.30	**3**
Gripper (20)	6	**8**	7	0.77	3.84	**8**
Hiking (20)	6	12	**13**	0.71	3.35	**13**
Logistics (63)	28	26	30	0.07	0.50	**39**
Miconic (150)	104	63	100	$<$ 0.01	0.19	**144**
Mprime (35)	26	**29**	**29**	0.32	1.87	27
Mystery (30)	16	**18**	**18**	0.06	2.58	17
Nomystery (20)	11	16	**20**	0.10	0.37	**20**
Openstacks (100)	18	49	45	0.55	7.10	**51**
Parcprinter (50)	28	20	32	0.13	0.86	**39**
Parking (40)	0	0	3	–	–	**8**
Pathways (30)	4	4	**5**	0.18	0.98	4
Pegsol (50)	6	46	**48**	0.30	4.47	**48**
Pipesw.-NT (50)	4	17	21	0.21	0.78	**23**
Pipesw.-T (50)	4	14	**16**	0.47	1.21	**16**
PSR (50)	48	**49**	**49**	1.16	2.02	**49**
Rovers (40)	4	8	**10**	0.25	0.76	7
Satellite (36)	4	6	**7**	0.03	0.55	**7**
Scanalyzer (50)	11	21	**23**	0.16	1.04	**23**
Sokoban (50)	44	41	**45**	0.53	1.50	**45**
Storage (30)	10	**16**	15	1.26	2.43	**16**
Tetris (17)	1	**9**	**9**	0.25	0.66	**9**
Tidybot (40)	3	26	**30**	1.45	2.24	22
TPP (30)	7	**11**	8	–	0.98	8
Transport (70)	7	24	**28**	0.68	2.07	25
Trucks (30)	3	10	10	0.17	0.70	**12**
Visitall (40)	12	13	13	0.16	1.07	**16**
Woodw. (50)	18	21	**35**	$<$ 0.01	0.24	32
Zenotravel (20)	8	12	**13**	0.17	0.80	**13**
aggregate (1637)	532	736	849	0.19	1.11	**957**

Table 1: Coverage for the basic online-refinement approach h^{on}, the baseline h^{off}, and online-refinement with restricted refinement time $h^{\text{on}}_{0.1}$ in the leftmost columns. The middle columns show the ratio of the expansions until the last f-layer is reached and search time for $h^{\text{on}}_{0.1}$ compared to h^{off}. The rightmost column shows coverage data for a state-of-the-art configuration of Cartesian abstraction heuristics.

This decrease in expansions comes with a trade-off in search time, as shown on the right side of Figure 6. While the maximum time for each expansion is consistently low in h^{off}, h^{on} can spend a lot of time in refinement and use up almost the entire search time to refine a few states. On commonly solved instances, the search time for h^{on} is 15 times larger than for h^{off} on average. Exceptions are Logistics, Miconic, and Woodworking, where h^{on} has lower search time, resulting in higher coverage in Miconic and Logistics.

Operation Time Distribution A significant fraction of the search time is used to improve the heuristic. This time is distributed over the three operations *refine*, *merge* and *reorder* (each including the recomputation of the cost partitioning and the updating of the stored h^* values in the abstraction), evaluating the *Bellman* equation, and updating the *open list*. Figure 7 shows the time distribution for each domain. The percentages are averaged over all instances of the

domain (including unsolved ones).

On average, about two thirds of the search time is used only to refine abstractions, but the variance is high and heavily depends on the domain. The extreme cases are Sokoban, where *no* time at all is spent on refinement, and Transport, with 99%. In Sokoban, there are applicable zero-cost actions in every state leading to states with equal heuristic value, so the Bellman equation is always satisfied. The impact of the *merge* operation depends on the number of goals facts. In domains with many goal facts, e.g. GED or Gripper, there are many small abstractions which need to be merged to enable further refinement operations. With about 1% the *reorder* time is negligible in all domains, which can be attributed to our simple ordering strategy. The time spent on the *Bellman* equation check highly depends on the branching factor of the domain as more heuristic values must be compared. In most domains, this accounts for less than 10% of the overall time, the only exceptions are Miconic (17%) and Sokoban (14%). The *open list* time is below 5% in almost all domains. This is mainly due to the fact that the open list stays relatively small due to the low number of expanded states.

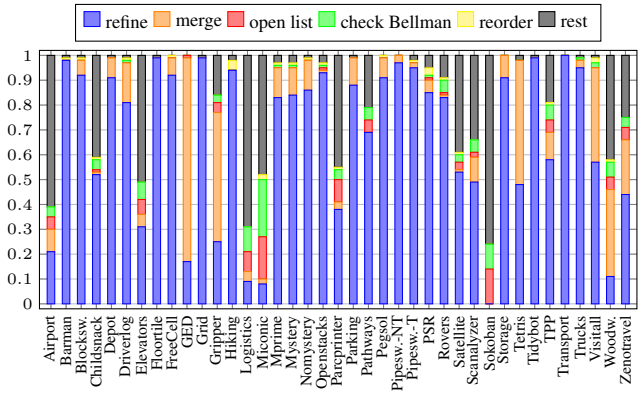

Figure 7: Average ratio between the time to improve the heuristic and the search time. The improvement time is split in five parts. Displayed is the average ratio per domain excluding tasks which have been solved in 0.01s or less.

Used Cost The OSCP algorithm is based on the assumption that there is still *unused cost*. In Figure 8 the average fraction of used cost per domain is shown, both for the initial abstractions and the final value when the instance is solved or the timeout is reached.

On average only 45% of the actions are used with non-zero cost in any abstraction in the beginning of the search, so typically there is enough unused cost that can be distributed among the abstractions in the OSCP. However, this number has a high variance depending on the domain, and there are some domains where almost the entire cost is used in the beginning already (e.g. Elevators and GED). In some domains, there is still a large amount of unused cost remaining, even at the end of the search (Mprime, Mystery, Tetris, Trucks).

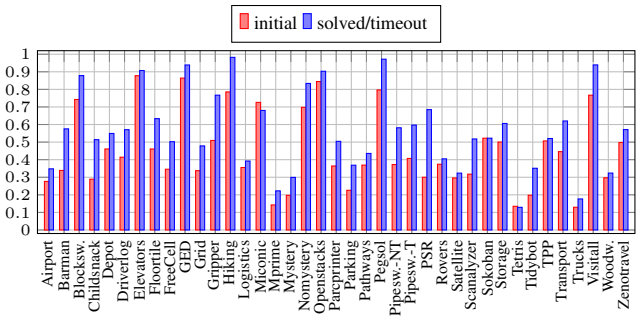

Figure 8: Fraction of the actions which are used in any abstraction with a non-zero cost. Displayed is the average over all orders per domain for the initial abstractions, and after the instance is solved or the time limit is reached.

Reducing the Refinement Overhead

If every local erroneous state is refined, the overhead introduced by the online-refinement procedure is very high. This leaves only little time remaining for the actual search process (c.f. the grey bars in Figure 7).

Addressing this issue, we experimented with additional configurations h^{on}_p where a parameter p is introduced that limits the time spent on refinement to a fixed fraction of the overall search time. The refinement process is only executed, if currently the fraction of the overall search time that is spent on refinement is below that threshold. The time spent to satisfy the Bellman equation in one state can still be very high. Therefore, we only perform at most one refinement operation in each state and do not merge any abstractions.

A graphical overview of the results for these configurations, using the values 0.01, 0.05, 0.10, $\ldots$, 0.40 for p, is shown in Figure 9. As we increase the refinement parameter p, the final abstraction size increases and with it the number of expansions needed to reach the final f-layer decreases. On the other hand, too much refinement overhead is also detrimental to the overall performance of this approach. The sweet spot lies at $p = 0.1$, where the highest overall coverage of 849 and lowest average search time is reached.

Compared to our base configuration h^{on}, the increase in coverage is consistent across almost all domains (the only exception is Miconic). Our configuration with restricted refinement diminishes the negative effect of the refinement overhead, and considerably improves over both h^{on} and h^{off}. It has a higher coverage than h^{off} in 26 domains, and only loses in 5 domains. On average, the number of expansions until the last f-layer is reduced by 81% (c.f. column "exp" in Table 1). However, the search time on commonly solved instances is often greater due to the added overhead of online refinement on instances where it is not required (c.f. column "time" in Table 1).

Comparison to State of the Art As a comparison to the state of the art in Cartesian abstractions, we compare our best performing configuration $h^{\text{on}}_{0.1}$ to an additive Cartesian abstractions heuristic $h^{\text{off}}_{\text{div}}$ that also uses landmark decomposition, and uses a diversified set of greedily instantiated orders for the saturated cost partitioning (Seipp 2017).

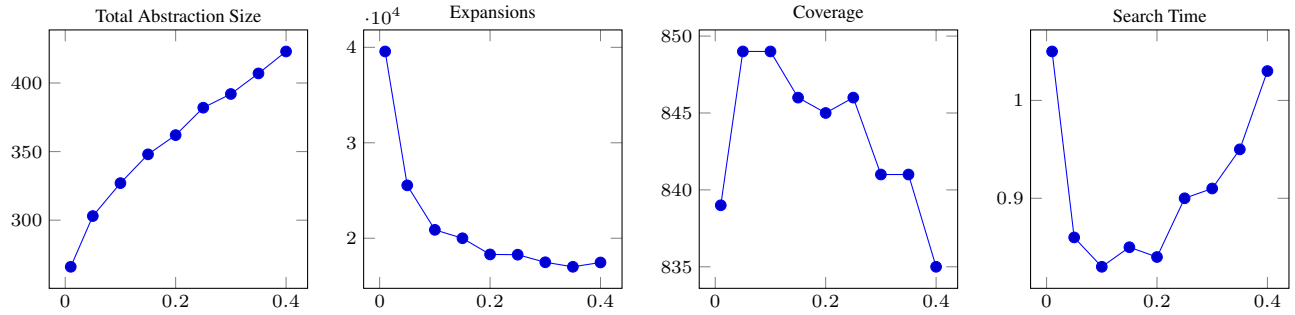

Figure 9: Results for h^{on}_p with p ranging from 0.01 to 0.4.

The results for $h^{\text{off}}_{\text{div}}$ are shown in the rightmost column in Table 1. In terms of overall coverage, $h^{\text{off}}_{\text{div}}$ beats our approach by a large margin, but mostly due to the big gaps in the FreeCell (+45) and Miconic (+44) domains. In 14 domains $h^{\text{off}}_{\text{div}}$ has higher coverage than $h^{\text{on}}_{0.1}$, while our approach works better in 10 domains. The biggest advantage for online refinement can be observed in Tidybot (+8).

Useful Splits In order to best evaluate the impact of the *useful split* check we use our h^{on}_p configuration with $p = 1$. Enabling this check can prevent the Bellman equation from being satisfied, so we can not use h^{on}.

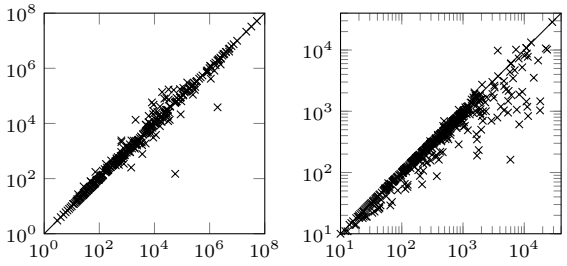

Figure 10: Left: Number of expansions until last f-layer. Right: Number of abstract states. The x- and y-axes correspond to without respectively with useful splits.

Figure 10 shows the number of expansions until the last f-layer is reached and the size of the abstractions. It shows that using the useful splits check can sometimes significantly reduce the size of the resulting abstractions while retaining very similar heuristic informativeness.

This improvement also translates to a higher coverage (679 with vs. 652 without useful splits). The domains benefitting the most are Pegsol (+8) and Scanalyzer (+4), but there are also domains where the coverage decreases, e.g. −3 in GED. For $h^{\text{on}}_{0.1}$, enabling the useful split check did not improve the overall results.

Online vs. Offline Refinement

Finally, we want to examine whether refinement based on the current search states leads to a more informed heuristic than doing refinement only in the initial state. While we already showed that h^{on} can reach the final f-layer with much fewer expansions than h^{off} (Figure 6), in that comparison the online-refined abstractions were allowed to grow a lot bigger than those generated offline.

In order to create a fair environment, both abstractions should have the same number of abstract states. The abstract state space size using goal abstractions and only offline refinement is often strictly limited. Hence, for this comparison, we use only a single abstraction containing all goal facts.

For this experiment, we first do a run with online refinement, starting from the trivial abstraction, and performing the online refinement procedure until each state satisfies the Bellman equation. After this run (when a solution is found or a time limit of 15 minutes is reached), we restart the search and use the resulting abstraction $\mathcal{A}^{\text{on}}$ without further online refinement. We compare this setting to a run with an offline-refined abstraction $\mathcal{A}^{\text{off}}$, using the number of abstract states of $\mathcal{A}^{\text{on}}$ as the abstract state space size bound during offline refinement.

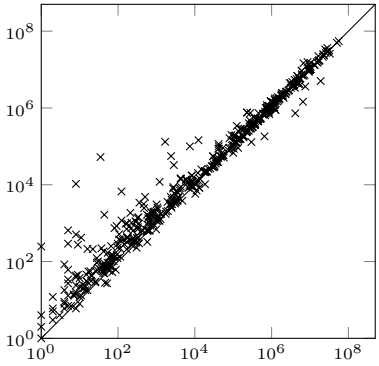

Figure 11: Comparison of the expansions until the last f-layer. The x- and y-axes correspond to $\mathcal{A}^{\text{on}}$ respectively $\mathcal{A}^{\text{off}}$.

Figure 11 compares the number of expansions until reaching the final f-layer for both resulting heuristics. The online-refined abstractions tend to need fewer expansions, on commonly solved instances the expansions are reduced to a factor of only 0.66 compared to $\mathcal{A}^{\text{off}}$. In only 4 out of 37 domains $\mathcal{A}^{\text{off}}$ is better, in all other domains using $\mathcal{A}^{\text{on}}$ results in a smaller search space. The greatest search space reduction can be observed in the domains Grid (0.12), Hiking (0.14), and Mprime (0.09). This also leads to a better overall coverage for $\mathcal{A}^{\text{on}}$ (690 vs. 679). Interestingly, the initial heuristic

value is often much lower with $\mathcal{A}^{\text{on}}$ compared to $\mathcal{A}^{\text{off}}$ (4.2 vs. 14.2, geometric mean over all instances). This shows that the fineness of the abstract state space is distributed more evenly in the online-refined abstraction.

Conclusion

We introduced a monotone converging online-refinement procedure for a set of additive Cartesian abstraction heuristics consisting of the three operations *refine*, *merge*, and *reorder*. Our results show that online refinement considerably improves the accuracy of the heuristic, but it has to be used carefully to avoid prohibitive overhead. When this overhead is bounded to a managable amount, our approach significantly improves over a heuristic using basic offline-refined abstractions, and even beats a heuristic using additional techniques such as landmark decomposition and greedily instantiated cost partitioning orders on many domains. In principle, these techniques could be combined with online refinement as well, so there is still more potential.

Another interesting direction for future work is devising more sophisticated strategies for refinement (i.e. which states to refine and how much). Similarly, different strategies to select which abstractions to merge could be tried, in particular for domains with many goals (and thus, many individual abstractions that can be merged).

Acknowledgments. This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation". We thank the anonymous reviewers, whose comments helped significantly to improve this paper.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In Felner, A., and Sturtevant, N. R., eds., *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SOCS'10)*. Stone Mountain, Atlanta, GA: AAAI Press.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *AI* 175(16-17):2075–2098.

Bäckström, C. 1995. Expressive equivalence of planning formalisms. *AI* 76(1–2):17–34.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. *AI* 221:73–114.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Proceedings of the 6th European Conference on Planning (ECP'01)*, 13–24. Springer-Verlag.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *JAIR* 22:279–318.

Fickert, M., and Hoffmann, J. 2017. Complete local search: Boosting hill-climbing through online heuristic-function refinement. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*. AAAI Press.

Fickert, M.; Hoffmann, J.; and Steinmetz, M. 2016. Combining the delete relaxation with critical-path heuristics: A direct characterization. *JAIR* 56(1):269–327.

Hart, P. E.; Nilsson, N. J.; and 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.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In Howe, A., and Holte, R. C., eds., *Proceedings of the 22nd National Conference of the American Association for Artificial Intelligence (AAAI'07)*, 1007–1012. Vancouver, BC, Canada: AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM* 61(3).

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Katz, M., and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *AI* 174(12–13):767–798.

Keyder, E.; Hoffmann, J.; and Haslum, P. 2014. Improving delete relaxation heuristics through explicitly represented conjunctions. *JAIR* 50:487–533.

Korf, R. E. 1990. Real-time heuristic search. *AI* 42(2-3):189–211.

Seipp, J., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 347–351. Rome, Italy: AAAI Press.

Seipp, J., and Helmert, M. 2014. Diverse and additive cartesian abstraction heuristics. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2017. Narrowing the gap between saturated and optimal cost partitioning for classical planning. In Singh, S., and Markovitch, S., eds., *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, 3651–3657. AAAI Press.

Seipp, J. 2012. Counterexample-guided abstraction refinement for classical planning. Master's thesis, University of Freiburg, Germany.

Seipp, J. 2017. Better orders for saturated cost partitioning in optimal classical planning. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SOCS'17)*. AAAI Press.

Wilt, C. M., and Ruml, W. 2013. Robust bidirectional search via heuristic improvement. In desJardins, M., and Littman, M., eds., *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI'13)*. Bellevue, WA, USA: AAAI Press.