Journal of Artificial Intelligence Research 62 (2018) 535–577 Submitted 10/17; published 07/18

Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning

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

Abstract

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

1. Introduction

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

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

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

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

One way of composing admissible heuristics is to use the maximum of their estimates. This combination is always admissible if the component heuristics are. In order to gain a more informed

©2018 AI Access Foundation. All rights reserved.

heuristic it would almost always be preferable to use the *sum* of the estimates, but this estimate is often not admissible. To remedy this problem, we can use *cost partitioning* (Katz & Domshlak, 2008) to ensure that the cost of each operator is distributed among the heuristics in a way that makes the sum of their estimates admissible.

To combine multiple Cartesian abstraction heuristics admissibly, we introduce a new cost partitioning algorithm, which we call *saturated cost partitioning*. Saturated cost partitioning opportunistically exploits situations where some operator costs can be lowered without affecting the quality of the heuristic. Given an ordered sequence of abstractions, the saturated cost partitioning algorithm computes the smallest cost function $cost'$ that is sufficient for achieving the same heuristic values under the first abstraction as with the full cost function $cost$. It then assigns cost function $cost'$ to this abstraction and continues the process with the remaining abstractions, using the leftover costs $cost - cost'$ as the new overall cost function. Compared to other cost partitioning algorithms for abstraction heuristics, one advantage of saturated cost partitioning is that the abstractions can be created one at a time. In particular, only a single abstraction needs to be held in memory simultaneously.

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

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

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

2. Background

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

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

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

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

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

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

Goal $s_\star = \{\textit{ball} \mapsto B\}$

Figure 1: Definition of the example Gripper task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ with a single ball, a gripper G and two rooms A and B.

Definition 1. Planning tasks.
A planning task *is a tuple* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$*, where:*

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

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

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

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

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

- $s_0 \in S(\Pi)$ *is the* initial state *and* $s_\star$ *is a partial state, called the* goal*.*

Figure 1 shows how the example Gripper task can be formalized using this definition. We will frequently use a tuple notation for states and write $\langle d_1, \dots, d_n \rangle$ to denote the state $\{v_1 \mapsto d_1, \dots, v_n \mapsto d_n\}$. (Facilitating such a notation is the main reason why we define $\mathcal{V}$ as a sequence rather than a set.)

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

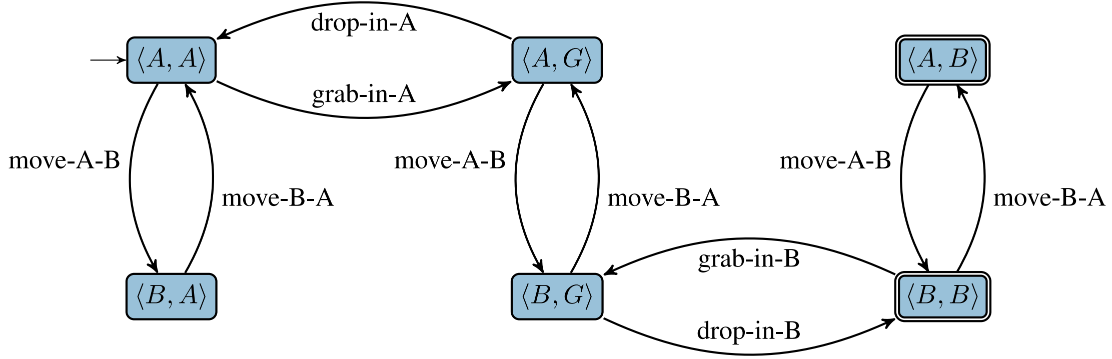


Figure 2: Transition system for the example Gripper task with a single ball, a gripper G and two rooms A and B.

Definition 2. Transition systems, regression, plans and traces.
A transition system $\mathcal{T} = \langle S, \mathcal{L}, T, s_0, S_\star \rangle$ *consists of a finite set of* states S*, a finite set of* labels $\mathcal{L}$*, a set of* transitions $T \subseteq S \times \mathcal{L} \times S$*, an* initial state $s_0 \in S$ *and a set of* goal states $S_\star \subseteq S$*. We write* $s \xrightarrow{l} s'$ *for* $\langle s, l, s' \rangle \in T$*, i.e., for a transition from state* $s \in S$ *to state* $s' \in S$ *with label* $l \in \mathcal{L}$*.*

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

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

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ induces a transition system $\mathcal{T}$ with states $\mathcal{S}(\Pi)$, initial state s_0, goal states $\{s \in \mathcal{S}(\Pi) \mid s_\star \subseteq s\}$ and transitions $\{s \xrightarrow{o} (s \oplus \mathit{eff}(o)) \mid s \in \mathcal{S}(\Pi), o \in \mathcal{O}, \mathit{pre}(o) \subseteq s\}$. Plans for $\mathcal{T}$ are also called plans for Π.

Associating a cost function *cost* with $\mathcal{T}$, which assigns the weight $\mathit{cost}(l) \in \mathbb{R}_0^+$ to all transitions with label l, turns the transition system into a weighted digraph (possibly with parallel arcs). A plan is *optimal* if the sum of assigned weights along the path is minimal. Optimal planning is the following problem: given a planning task Π with initial state s_0 and operator cost function *cost*, find an optimal plan starting in s_0 in the transition system induced by Π and weighted by *cost*, or prove that no plan starting in s_0 exists. Figure 2 shows the (unweighted) transition system induced by the Gripper example. We follow the common convention and always mark the initial state with an unlabeled incoming edge and goal states with double borders.

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

Definition 3. Abstractions.
Let Π *be a planning task inducing the transition system* $\langle S, \mathcal{L}, T, s_0, S_\star \rangle$*.*

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

mapping s *to* $[s]_\sim$ *is called the* abstraction function. *We omit the subscript* $\sim$ *where clear from context.*

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

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

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

Abstraction preserves paths in the transition system and can therefore be used to define admissible and consistent heuristics for planning (e.g., Katz & Domshlak, 2008; Helmert, Haslum, & Hoffmann, 2007; Katz & Domshlak, 2010). Specifically, $h^\sim(s, \mathit{cost})$ is defined as the cost of an optimal plan starting from $[s]$ in the abstract transition system (weighted by cost function *cost*) or ∞ if no plan starting from $[s]$ exists. Practically useful abstractions should be efficiently computable and give rise to informative heuristics. These are conflicting objectives.

3. Cartesian Abstractions

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

During abstraction refinement, we frequently need to check whether a given operator induces a transition between two given abstract states. We now show that this check is expensive for merge-and-shrink abstractions. Let BDD_A and BDD_B be (reduced ordered) BDDs (Bryant, 1986) over the binary variables $\{v_1, \dots, v_n\}$ such that BDD_A represents the set of truth assignments A and BDD_B represents the set of truth assignments B. Consider the planning task with variables $\langle v_0, v_1, \dots, v_n \rangle$, $\mathit{dom}(v_0) = \{0, 1\}$ and a single operator o with $\mathit{pre}(o) = \{v_0 \mapsto 0\}$ and $\mathit{eff}(o) = \{v_0 \mapsto 1\}$. Let $a = \{x \cup \{v_0 \mapsto 0\} \mid x \in A\}$ and $b = \{y \cup \{v_0 \mapsto 1\} \mid y \in B\}$ be two abstract states in a merge-and-shrink abstraction. Then A and B have a non-empty intersection iff o induces a transition between a and b.

Due to the known relationships between merge-and-shrink abstractions and BDDs, we can construct merge-and-shrink representations of a and b with identical representation size (modulo small constants) to BDD_A and BDD_B if we use a linear merge strategy based on the same variable order as the one used for the BDDs, with v_0 at the front (Helmert, Röger, & Sievers, 2015). The best algorithms known for testing if BDD_A and BDD_B have a non-empty intersection takes time $O(\|\mathrm{BDD}_A\| \cdot \|\mathrm{BDD}_B\|)$, where $\|X\|$ is the number of nodes of X. If the transition check needed less time, then testing whether two BDDs have a non-empty intersection could also be done faster, but no algorithm with a better runtime is known. In a typical merge-and-shrink abstraction, the

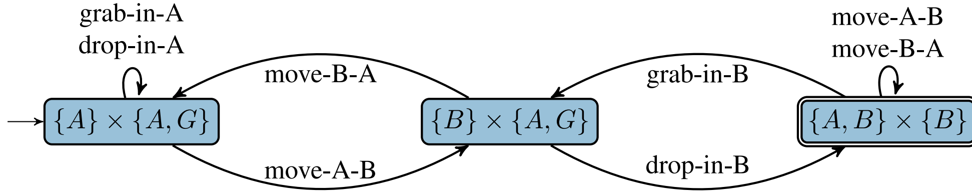


Figure 3: Example Cartesian abstraction of the Gripper example. In the left and center state we know where the robot is, but not whether its gripper holds the ball, whereas in the abstract goal state we ignore the position of the robot.

BDDs involved can have 10^4 or more nodes (e.g., Sievers, 2017), so a single transition test can take 10^8 or more computation steps. This is prohibitively expensive for the refinement approach we consider in this paper, which can perform millions of transition tests.

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

Definition 4. Cartesian sets and Cartesian abstractions.
A set of states for a planning task with variables $\langle v_1, \ldots, v_n \rangle$ *is called* Cartesian *if it is of the form* $A_1 \times A_2 \times \ldots \times A_n$*, where* $A_i \subseteq dom(v_i)$ *for all* $1 \leq i \leq n$*.*

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

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

Figure 3 shows an example Cartesian abstraction for the Gripper task. The Cartesian sets $\{A\} \times \{A, G\}$, $\{B\} \times \{A, G\}$ and $\{A, B\} \times \{B\}$ are the states in the abstract transition system. The heuristic $h^{\sim}$ maps each state to the respective abstract goal distance (2, 1 or 0 — under the unit cost function).

The name "Cartesian abstraction" was coined in the model-checking literature by Ball, Podelski, and Rajamani (2001) for a concept essentially equivalent to Definition 4. (Direct comparisons are difficult due to different state models.) We discuss their work in Section 7.

Cartesian sets naturally arise in planning tasks. In particular, for every partial state s, the set of states that agree with s (i.e., all states s' with $s'[v] = s[v]$ for all $v \in vars(s)$) is Cartesian.

Definition 5. Cartesian sets for partial states.
Let s *be a partial state of a planning task with variables* $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$*. The Cartesian set induced by* s *is defined as*

$$Cartesian(s) = A_1 \times \ldots \times A_n, \quad where \quad A_i = \begin{cases} \{s[v_i]\} & \text{if } v_i \in vars(s) \\ dom(v_i) & \text{otherwise} \end{cases}.$$

Clearly, *Cartesian*(s) contains exactly the states that agree with s. It follows that many important sets of states in a planning task, such as the set of goal states or the set of states in which a given operator is applicable, are Cartesian.

Cartesian abstractions form a fairly general class; e.g., they include *projections* (the abstractions underlying pattern databases) and *domain abstractions* (Hernádvölgyi & Holte, 2000) as special

cases. A projection is an abstraction that only considers the information from a subset $P \subseteq \mathcal{V}$ of the state variables, called the *pattern*. For a state s, let $s|_P$ be the partial state defined on P with $s|_P[v] = s[v]$ for all $v \in P$. Two states s and s' are equivalent under the projection to P iff $s|_P = s'|_P$. It is easy to see that this is a Cartesian abstraction, where the equivalence class of state s is $[s]_\sim = \mathit{Cartesian}(s|_P)$.

A domain abstraction is an abstraction that is defined in terms of local equivalence relations, one for each state variable. Consider a planning task with state variables $\langle v_1, \dots, v_n \rangle$ and equivalence relations $\sim_1, \dots, \sim_n$, where $\sim_i$ is an equivalence relation on $\mathit{dom}(v_i)$. The domain abstraction induced by $\sim_1, \dots, \sim_n$ is the abstraction relation $\sim$ defined as $s \sim s'$ iff $s[v_i] \sim_i s'[v_i]$ for all $1 \leq i \leq n$. It is easy to see that domain abstractions are a proper generalization of projections: projections arise in the special case where each local abstraction is either the identity relation ($d \sim_i d'$ iff $d = d'$) or the universal relation ($d \sim_i d'$ for all d, d'). It is also easy to see that domain abstractions are Cartesian abstractions, where each equivalence class is a Cartesian product of equivalence classes of the local equivalence relations. Specifically, we have $[s]_\sim = [s[v_1]]_{\sim_1} \times \dots \times [s[v_n]]_{\sim_n}$, and hence every equivalence class of $\sim$ is a Cartesian set.

Unlike projections and domain abstractions, general Cartesian abstractions can have arbitrarily different levels of granularity in different parts of the abstract state space. One abstract state might correspond to a single concrete state while another abstract state corresponds to half of the states of the task.

We illustrate the relationships between different classes of abstractions with example abstractions of our Gripper task. Figure 4a shows the abstract transition system induced by the projection to the pattern $\{\mathit{robot}\}$. Clearly, this abstraction is also a domain abstraction, Cartesian abstraction and merge-and-shrink abstraction. When we additionally partition the domain for variable *ball* into the groups $\{A\}$ and $\{B, G\}$, the resulting abstraction is not a projection anymore (Figure 4b). It is however still a domain abstraction. A further split of state $\{B\} \times \{B, G\}$ into the two states $\{B\} \times \{B\}$ and $\{B\} \times \{G\}$ yields the Cartesian abstraction in Figure 4c. Since not all domains are split equally for all states, it is not a domain abstraction anymore. Combining the states $\{A\} \times \{B, G\}$ and $\{B\} \times \{A\}$ results in the transition system in Figure 4d. This abstraction is not Cartesian anymore, but can be expressed in the merge-and-shrink formalism.

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

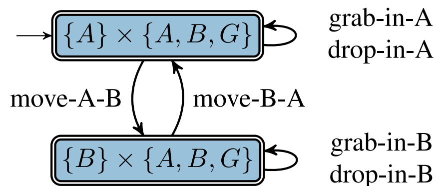
The following theorem collects a number of properties that make Cartesian sets interesting for CEGAR in planning. We remind the reader that a *partition* of a set X is a collection of subsets $X_1, \dots, X_k \subseteq X$ that are pairwise disjoint and jointly exhaustive, i.e., $X_i \cap X_j = \emptyset$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

Theorem 1. Properties of Cartesian sets.
Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task.

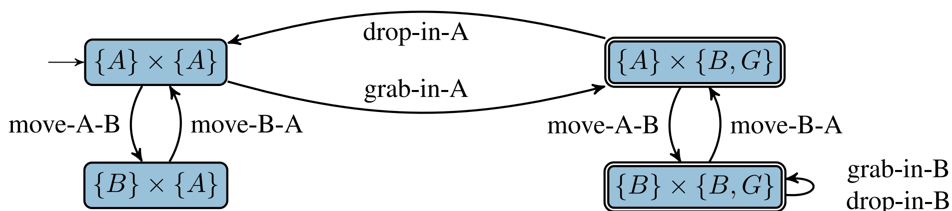
(P1) *The set of goal states of Π is Cartesian.*

(P2) *For all operators $o \in \mathcal{O}$, the set of states in which o is applicable is Cartesian.*

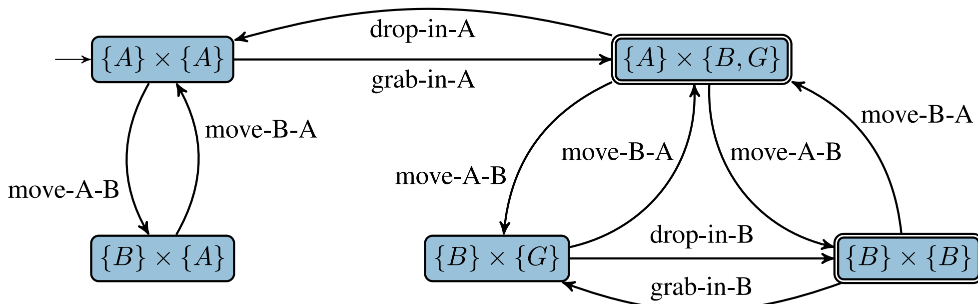
(P3) *The intersection of Cartesian sets is Cartesian.*



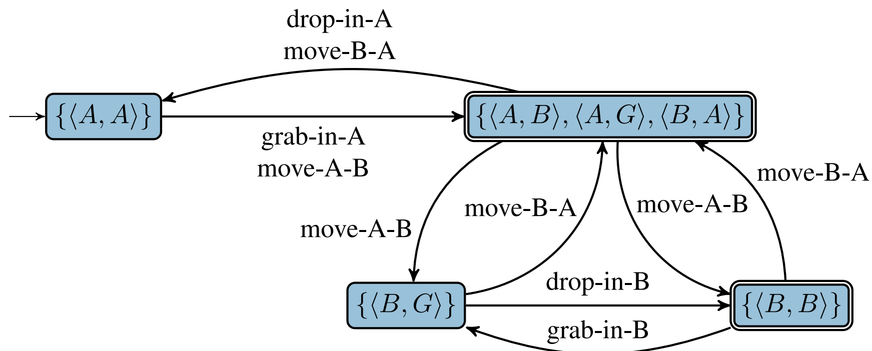
(a) Projection.



(b) Domain abstraction.



(c) Cartesian abstraction.



(d) Merge-and-shrink abstraction.

Figure 4: Example abstractions of the Gripper task for different classes of abstractions. The captions state the most specific class each abstraction belongs to.

(P4) *For all operators $o \in \mathcal{O}$, the regression of a Cartesian set with respect to o is Cartesian.*

(P5) *If $b \subseteq a$ and $c \subseteq a$ are disjoint Cartesian subsets of the Cartesian set a, then a can be partitioned into Cartesian sets d and e with $b \subseteq d$ and $c \subseteq e$.*

(P6) *If $c \subseteq a$ is a Cartesian subset of the Cartesian set a and $s \in a \setminus c$, then a can be partitioned into Cartesian sets d and e with $s \in d$ and $c \subseteq e$.*

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

(P1) The set of goal states is $\mathit{Cartesian}(s_\star)$.

(P2) Consider any operator $o \in \mathcal{O}$. The set of states where o is applicable is $\mathit{Cartesian}(\mathit{pre}(o))$.

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

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

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

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

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

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

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

□

4. Abstraction Refinement

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

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

```
 1: function CEGAR()
 2:   T' ← TRIVIALABSTRACTION()
 3:   while not TERMINATIONCONDITION() do
 4:     τ' ← FINDOPTIMALTRACE(T')
 5:     if τ' is "no trace" then
 6:       return task is unsolvable
 7:     φ ← FINDFLAW(τ')
 8:     if φ is "no flaw" then
 9:       return plan extracted from τ'
10:     T' ← REFINE(T', φ)
11:   return T'
```

4.1 Refinement Algorithm

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

4.1.1 Main Loop

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

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

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

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

4.1.2 Trace Verification

The FINDFLAW function in Algorithm 2 attempts to convert the abstract goal trace τ' into a *concrete* goal trace with the same label sequence. If this succeeds, it returns the special value "no flaw".

Algorithm 2 Trace verification. Tries to convert a given abstract goal trace τ' into a concrete goal trace. If the conversion fails, returns the first encountered "flaw", i.e., a pair of a concrete state s and a Cartesian set $c \subseteq [s]$ such that converting the trace failed because $s \notin c$. If the conversion succeeds, the function returns "no flaw".

1: **function** FINDFLAW(τ')
2: $s \leftarrow s_0$
3: **for each** $(a \xrightarrow{o} b) \in \tau'$ **do**
4: **if** o is not applicable in s **then**
5: **return** $\langle s, [s] \cap \mathit{Cartesian}(\mathit{pre}(o)) \rangle$
6: **if** b does not include $s[\![o]\!]$ **then**
7: **return** $\langle s, [s] \cap \mathit{regr}(b, o) \rangle$
8: $s \leftarrow s[\![o]\!]$
9: **if** s is not a goal state **then**
10: **return** $\langle s, [s] \cap \mathit{Cartesian}(s_\star) \rangle$ with
11: **return** "no flaw"

Otherwise, it returns a *flaw*, which we define as a pair $\langle s, c \rangle$ where s is a concrete state and $c \subseteq [s]$ is a Cartesian set such that the conversion of the abstract trace failed because $s \notin c$. Let $\tau' = \langle a_0 \xrightarrow{o_1} a_1, a_1 \xrightarrow{o_2} a_2, \ldots, a_{k-1} \xrightarrow{o_k} a_k \rangle$. We attempt to find a concrete trace $\langle s_0 \xrightarrow{o_1} s_1, s_1 \xrightarrow{o_2} s_2, \ldots, s_{k-1} \xrightarrow{o_k} s_k \rangle$ where s_0 is the concrete initial state, $[s_i] = a_i$ for all $0 \leq i \leq k$, and s_k is a concrete goal state. The conversion procedure starts from the initial state $s = s_0$ of the concrete task (line 2) and iteratively applies the next operator o in τ' until one of the following situations is encountered:

1. Operator o is not applicable in concrete state s (line 4). In this case, the flaw is $\langle s, c \rangle$, where c is the set of concrete states in $[s]$ in which o is applicable.

2. For abstract and concrete transitions $a \xrightarrow{o} b$ and $s \xrightarrow{o} s[\![o]\!]$ we have that $[s] = a$ but $[s[\![o]\!]] \neq b$, i.e., the concrete and abstract traces diverge (line 6). This can happen because abstract transition systems are not necessarily deterministic, and therefore the same abstract state can have multiple outgoing transitions with the same label. In this case, the flaw is $\langle s, c \rangle$ where c is the set of concrete states in $[s]$ from which we can reach b by applying o.

3. The concrete goal trace has been completed, but the last concrete state s is not a goal state (line 9). In this case, the flaw is $\langle s, c \rangle$ where c is the set of concrete goal states in $[s]$.

Properties P1, P2, P3 and P4 from Theorem 1 entail that c is a Cartesian set in all three cases. If none of these conditions occur, τ' corresponds to a concrete plan, so there is no flaw (line 11).

4.1.3 REFINEMENT

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

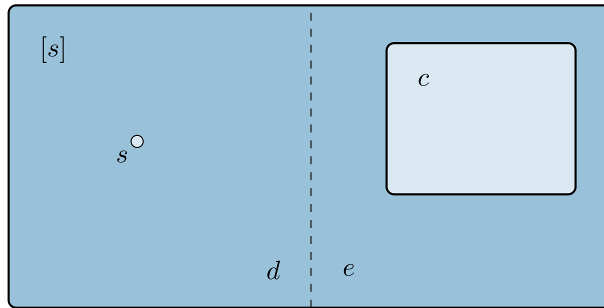


Figure 5: Illustration of how an abstract state $[s]$ must be split into two new abstract states d and e for a flaw $\langle s, c \rangle$.

Property P6 from Theorem 1 guarantees that it is always possible to partition the abstract state $[s]$ into two Cartesian sets d and e that separate s from c (i.e., $s \in d$ and $c \subseteq e$). This is illustrated in Figure 5. For the respective cases this split ensures that

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

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

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

1. In the first case, c is the set of concrete states in $[s]$ in which o is applicable. Because $\mathcal{T}'$ is an induced abstraction and has a transition starting in $[s]$ labeled with o, c cannot be empty.
2. In the second case, c is the set of concrete states in $[s]$ from which we can reach b by applying o. Because $\mathcal{T}'$ is an induced abstraction and has the transition $[s] \xrightarrow{o} b$, c cannot be empty.
3. In the third case, c is the set of concrete goal states in $[s]$. Because $\mathcal{T}'$ is an induced abstraction and $[s]$ is an abstract goal state, c cannot be empty.

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

The REFINE function in Algorithm 3 shows the refinement process. It splits $[s]$ into two new abstract states d and e as explained above and updates the abstract transition system by replacing $[s]$

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

```
 1: function REFINE(𝒯′, φ)
 2:     ⟨S′, ℒ′, T′, [s₀], S′⋆⟩ ← 𝒯′
 3:     ⟨s, c⟩ ← φ
 4:     ⟨d, e⟩ ← SPLIT(s, c)
 5:     S″ ← (S′ \ {[s]}) ∪ {d, e}
 6:     T″ ← REWIRETRANSITIONS(T′, [s], d, e)
 7:     if [s] = [s₀] then
 8:         a₀ ← d
 9:     else
10:         a₀ ← [s₀]
11:     if [s] ∈ S′⋆ then
12:         S″⋆ ← (S′⋆ \ {[s]}) ∪ {e}
13:     else
14:         S″⋆ ← S′⋆
15:     return ⟨S″, ℒ′, T″, a₀, S″⋆⟩
```

Algorithm 4 Transition rewiring. Given a set of abstract transitions T', an abstract state $[s]$ that is currently being split, and the two resulting new abstract states d and e, calculate the new set of induced transitions.

```
 1: function REWIRETRANSITIONS(T′, [s], d, e)
 2:     T″ ← T′
 3:     for each a -o-> b ∈ T″ with a = [s] do
 4:         T″ ← T″ \ {a -o-> b}
 5:         if CHECKTRANSITION(d, o, b) then
 6:             T″ ← T″ ∪ {d -o-> b}
 7:         if CHECKTRANSITION(e, o, b) then
 8:             T″ ← T″ ∪ {e -o-> b}
 9:     for each a -o-> b ∈ T″ with b = [s] do
10:         T″ ← T″ \ {a -o-> b}
11:         if CHECKTRANSITION(a, o, d) then
12:             T″ ← T″ ∪ {a -o-> d}
13:         if CHECKTRANSITION(a, o, e) then
14:             T″ ← T″ ∪ {a -o-> e}
15:     return T″
```

with d and e. In general, there can be many possible splits that separate s from c, and hence many possible choices of d and e. We discuss this choice in Section 4.3.

Next, the REFINE function "rewires" the transitions with the REWIRETRANSITIONS function shown in Algorithm 4. Since only a single state is split into two new states, the rewiring procedure only needs to make local changes to the transition system. Concretely, it needs to decide for each

Algorithm 5 Transition check. Returns true iff operator o induces a transition between abstract states a and b.

```
1: function CHECKTRANSITION(a, o, b)
2:   for each v ∈ V do
3:     if v ∈ vars(pre(o)) and pre(o)[v] ∉ dom(v, a) then
4:       return false
5:     if v ∈ vars(post(o)) and post(o)[v] ∉ dom(v, b) then
6:       return false
7:     if v ∉ vars(post(o)) and dom(v, a) ∩ dom(v, b) = ∅ then
8:       return false
9:   return true
```

incoming and outgoing transition of $[s]$, including self-loops, whether the transition needs to be rewired from/to d, from/to e, or both. This check is done by CHECKTRANSITION in Algorithm 5. (We discuss more efficient alternatives to Algorithms 4 and 5 in Section 4.3.)

The last step of the REFINE function is to update the abstract initial state and abstract goal states, if necessary. Due to the way we split $[s]$ into d and e, e can never be the abstract initial state and d can never be an abstract goal state.

It is easy to verify that REFINE preserves the abstraction property (see Definition 3): if $\mathcal{T}'$ is an abstraction of the concrete transition system, then the refined abstract transition system $\mathcal{T}''$ is also an abstraction of the concrete transition system, with one more abstract state than $\mathcal{T}'$. Moreover, it is also easy to verify that REFINE preserves inducedness: if $\mathcal{T}'$ is an induced abstraction, then so is $\mathcal{T}''$. Because $\mathcal{T}''$ is a refinement of $\mathcal{T}'$ in the sense that every plan of $\mathcal{T}''$ is a plan of $\mathcal{T}'$ (but not vice versa), all heuristic values based on $\mathcal{T}''$ are at least as large as the corresponding heuristic values based on $\mathcal{T}'$.

4.2 Example CEGAR Abstraction

Figure 6a shows the initial abstraction for our Gripper example task Π. If we use this abstraction to define a heuristic h_1, we obtain $h_1(s_0) = 0$ because $[s_0]$ is an abstract goal state. The empty abstract goal trace $\langle\rangle$ fails to solve Π because s_0 does not satisfy the goal. Therefore, REFINE splits $[s_0]$ based on the goal variable, leading to the finer abstraction in Figure 6b. The heuristic h_2 induced by this abstraction yields $h_2(s_0) = 1$ because ⟨drop-in-B⟩, the optimal abstract plan for $[s_0] = \{A, B\} \times \{A, G\}$, has a cost of 1.

The abstract goal trace $\langle(\{A, B\} \times \{A, G\}) \xrightarrow{\text{drop-in-B}} (\{A, B\} \times \{B\})\rangle$ corresponding to this abstract plan in Figure 6b does not solve Π because two preconditions of drop-in-B are violated in s_0: *ball* $\mapsto G$ and *robot* $\mapsto B$. We assume that REFINE performs a split based on variable *robot* (a split based on *ball* is also possible), which leads to Figure 6c. The abstraction heuristic h_3 for this abstraction produces the estimate $h_3(s_0) = 2$.

A further refinement step, splitting on *ball*, yields the system in Figure 6d with the optimal abstract goal trace $\langle(\{A\} \times \{A, G\}) \xrightarrow{\text{move-A-B}} (\{B\} \times \{G\}), (\{B\} \times \{G\}) \xrightarrow{\text{drop-in-B}} (\{A, B\} \times \{B\})\rangle$ and the heuristic estimate $h_4(s_0) = 2$. The first operator is applicable in s_0 and takes us into state s_1 with $s_1(\textit{robot}) = B$ and $s_1(\textit{ball}) = A$, but the second abstract state $a_1 = \{B\} \times \{G\}$ of the goal trace does not abstract s_1, i.e., the abstract and concrete plans diverge. Regression from a_1

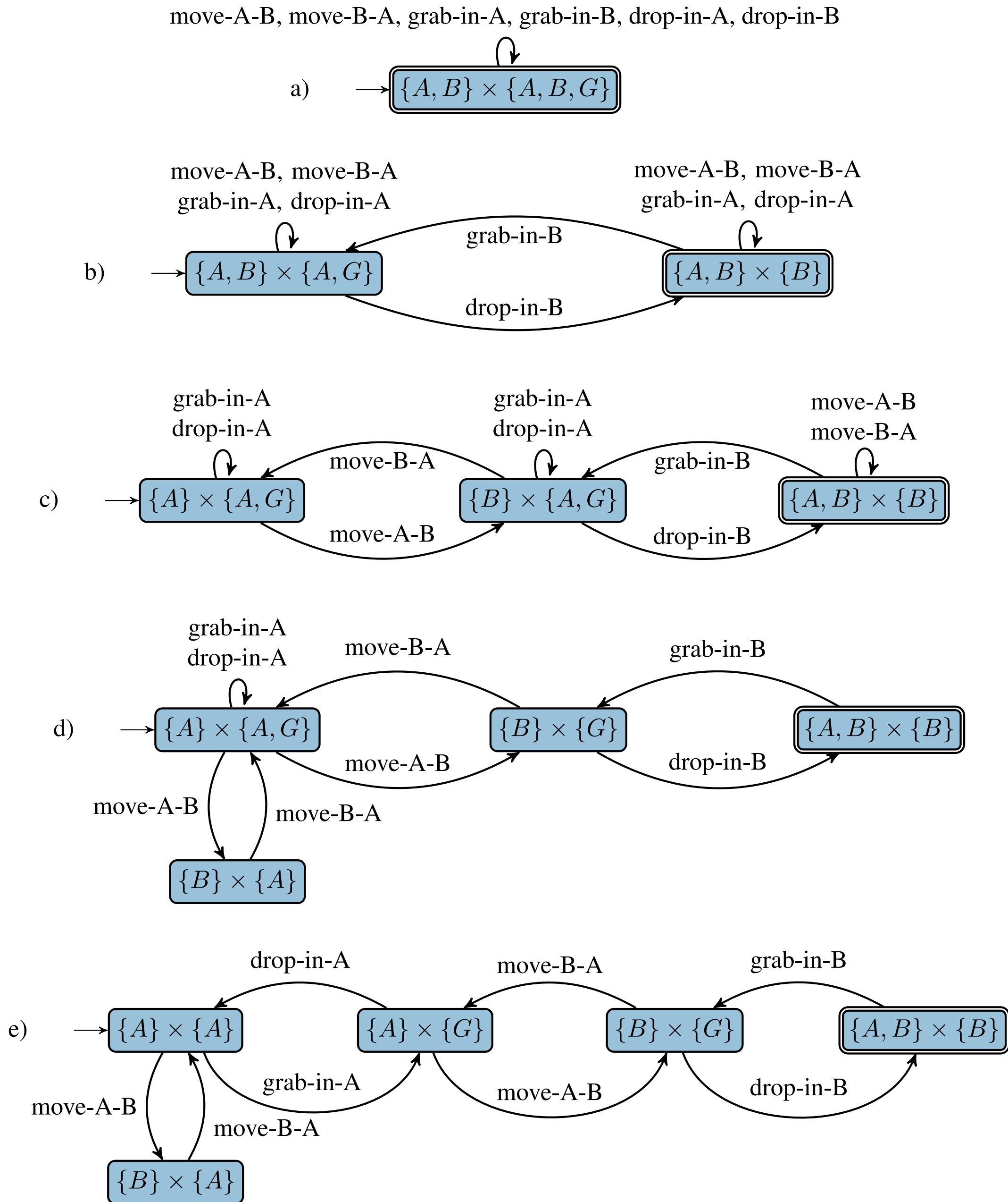


Figure 6: Refining the example abstraction.

with respect to move-A-B yields the Cartesian set $c = \{A\} \times \{G\}$, and hence REFINE must split the abstract initial state $[s_0]$ into two new states d and e in such a way that $s_0 \in d$ and $c \subseteq e$.

The result of this refinement is shown in Figure 6e. Under this abstraction we can define a heuristic h_5 which yields $h_5(s_0) = 3$. The optimal plan for this abstraction is a valid concrete plan, so we stop refining.

4.3 Implementation

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

4.3.1 REPRESENTATION OF CARTESIAN SETS

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

4.3.2 REFINEMENT STRATEGY

As noted above, when handling the flaw $\langle s, c \rangle$, we have to partition $[s]$ into two new abstract states d and e with $s \in d$ and $c \subseteq e$. Due to the nature of Cartesian abstractions, the only way of splitting $[s]$ into two Cartesian sets d and e is to choose a single variable v with $s[v] \notin \mathit{dom}(v, c)$ and partition $\mathit{dom}(v, [s])$ into $\mathit{dom}(v, d)$ and $\mathit{dom}(v, e)$ (cf. the proof of property P6 in Theorem 1). All other abstract domains must remain the same as in $[s]$, i.e., $\mathit{dom}(v', d) = \mathit{dom}(v', e) = \mathit{dom}(v', [s])$ for all $v' \in \mathcal{V} \setminus \{v\}$. The question is how to choose v and how to partition $\mathit{dom}(v, [s])$.

Let us first consider the question of selecting a variable v on which to split $[s]$. After some preliminary experiments we chose the following variable-selection strategy, which we call "max-refined". Out of the candidate variables, it selects the one that has already been refined the most in $[s]$, i.e., the variable v that minimizes the fraction $\frac{|\mathit{dom}(v,[s])|}{|\mathit{dom}(v)|}$ among all variables for which splits are feasible. We break ties by choosing the variable with the smallest index. This strategy clearly outperforms picking splits randomly and its inverse strategy, i.e., "min-refined".

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

4.3.3 LOOKUP FUNCTION

For the heuristic to be efficiently computable, we must be able to retrieve heuristic values very fast. The most critical operation here is computing the abstract state $[s]$ given a concrete state s. To make this operation efficient, we store a *refinement hierarchy* of split abstract states. This hierarchy is a binary tree of abstract states whose root is the first abstract state that was split, i.e., the only abstract state in the trivial abstract transition system. Whenever a state is split, the two resulting states become its child nodes. The leaves of the refinement hierarchy are the states in the

Algorithm 6 Abstract state lookup function. Given the concrete state s, return the abstract state $[s]$ by traversing the refinement hierarchy.

```
1:  function GETABSTRACTSTATE(s)
2:      a ← root of the refinement hierarchy
3:      while HASCHILDREN(a) do
4:          b, c ← GETCHILDREN(a)
5:          v ← GETSPLITVARIABLE(a)
6:          if s[v] ∈ dom(v, b) then
7:              a ← b
8:          else
9:              a ← c
10:     return a
```

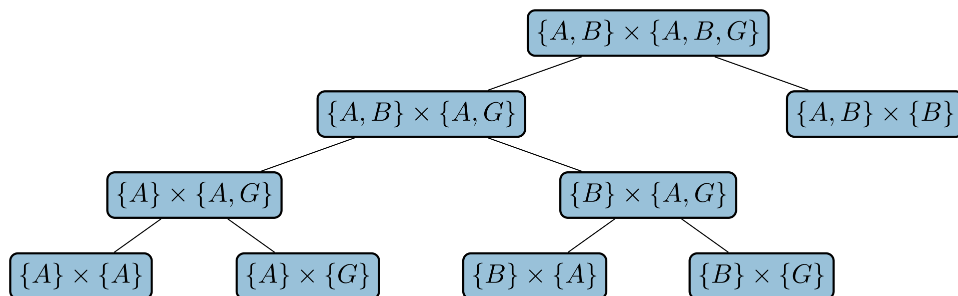


Figure 7: Refinement hierarchy for the final abstraction of the Gripper example from Figure 6e.

final abstraction. In addition to the child nodes we also store the variable on which each abstract state was split. Figure 7 shows an example refinement hierarchy and GETABSTRACTSTATE in Algorithm 6 demonstrates how we use refinement hierarchies for looking up abstract states. We analyze the runtime complexity of this operation below.

4.3.4 A* SEARCH

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

4.3.5 SELF-LOOPS

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

4.3.6 REWIRING TRANSITIONS EFFICIENTLY

Since we have to rewire numerous transitions during each refinement, we need to make this operation as fast as possible. Algorithm 4 shows a straightforward but inefficient implementation, which iterates over all transitions, determines which ones are affected by the current refinement step, and then uses Algorithm 5 to decide whether a given hypothetical transition is part of the abstract transition system.

The actual implementation does not iterate over all transitions, but maintains the incoming and outgoing transitions of each abstract state for direct access. The transitions are updated in-place, not copied and modified as in Algorithm 4.

Moreover, the individual transition checks can be made more efficient by exploiting the following observation: whenever we check for the existence of a transition $a'' \xrightarrow{o} b''$ in the refined transition system $\mathcal{T}''$, this check is triggered by an existing transition $a' \xrightarrow{o} b'$ in the current transition system $\mathcal{T}'$, where $a'' = a'$ or a'' is obtained from a' by splitting based on the variable v, and similarly $b'' = b'$ or b'' is obtained from b' by splitting based on the variable v. This means that a'' agrees with a' on all variables other than v, and b'' agrees with b' on all variables other than v. Because $\mathcal{T}'$ is an induced transition system with the transition $a' \xrightarrow{o} b'$, we know that none of the variables $v' \neq v$ can cause the transition check to fail. Therefore, it is sufficient to only consider variable v to determine whether the refined transition exists. This greatly speeds up the refinement loop. In our implementation we split the generic transition check into three specialized procedures, one for each type of transition: REWIREINCOMINGTRANSITION (Algorithm 9), REWIREOUTGOINGTRANSITION (Algorithm 10) and REWIRESELFLOOP (Algorithm 11). They are shown in Appendix A.

4.4 Theoretical Runtime Analysis

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

Theorem 2. Runtimes of operations on Cartesian sets.
If k is the number of atoms in a planning task Π, the following functions are computable in time $O(k)$ for Cartesian abstractions $\sim$:

(R1) *Compute the intersection of two Cartesian sets.*

(R2) *Compute the regression of operator o over a Cartesian set.*

(R3) *Given $s \in S(\Pi)$, compute $[s]_\sim$ and $h^\sim(s)$ (after abstract goal distances have been precomputed).*

(R4) *Given Cartesian sets a and b and operator o, decide if $a \xrightarrow{o} b$ is an abstract transition.*

(R5) *Given state $s \in S(\Pi)$ and Cartesian set a, decide whether a contains s.*

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

(R1) Intersecting two Cartesian sets a and b with $a = A_1 \times \dots \times A_n$ and $b = B_1 \times \dots \times B_n$ by intersecting the bit vectors takes time $O(k)$.

(R2) The proof of property P4 of Theorem 1 on page 541 shows how the regression is computed. All membership tests can be computed in $O(1)$. Therefore, the worst-case time complexity is $O(k)$.

(R3) Algorithm 6 on page 551 shows how we compute the abstract state corresponding to a given concrete state. Its runtime is determined by the number of iterations and the time for each iteration. There can be at most k iterations, since in the worst case we split off one of the k atoms from the same abstract state k times. Because the set membership test in line 6 runs in constant time, computing $[s]_\sim$ given $s \in S(\Pi)$ runs in time $O(k)$. Once we have computed the abstract state, retrieving the heuristic value is a constant-time lookup operation. Therefore, also computing $h^\sim(s)$ given $s \in S(\Pi)$ takes time $O(k)$.

(R4) Algorithm 5 on page 548 shows the general procedure for checking if a transition exists between two states. The runtime of the third case in the loop dominates the runtimes of the other two because it involves the intersection of domains instead of a simple membership test. One intersection for variable $v \in \mathcal{V}$ runs in time $O(|dom(v)|)$. In the worst case an intersection is performed for each variable $v \in \mathcal{V}$, and thus the whole algorithm has the asymptotic runtime $O(\sum_{i=1}^{n} |dom(v_i)|) = O(k)$.

(R5) Let $b = \{s[v_1]\} \times \ldots \times \{s[v_n]\}$ be the Cartesian set that only contains s. Constructing b takes time $O(k)$ since we represent it as a bit vector of length k. The intersection $a \cap b$ is non-empty iff a contains s. Using property R1 we can see that the whole operation runs in time $O(k)$.

□

5. Multiple Abstractions

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

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

5.1 Saturated Cost Partitioning

When using multiple admissible heuristics, we want them to be *additive* in order to obtain a more informed overall estimate, i.e., we want the sum of their individual estimates to be admissible. One way to achieve this is to use *cost partitioning* (Katz & Domshlak, 2008), dividing operator costs among multiple cost functions. Traditionally, these cost functions were required to assign non-negative costs to operators, but as Pommerening, Helmert, Röger, and Seipp (2015) showed, this restriction is unnecessary.

Definition 6. Cost partitioning.
A cost partitioning *for a planning task with operators* $\mathcal{O}$ *and cost function cost is a sequence* $cost_1, \ldots, cost_n$ *of cost functions* $cost_i : \mathcal{O} \to \mathbb{R}$ *such that* $\sum_{1 \leq i \leq n} cost_i(o) \leq cost(o)$ *for all operators* $o \in \mathcal{O}$.

Cost partitioning can be used to enforce additivity of a family of heuristics $h_1, \ldots, h_n$. Each heuristic h_i is evaluated according to its own "private" operator cost function $cost_i$. If each h_i is admissible (i.e., $h_i(s, cost) \leq h^*(s, cost)$ for all states s and cost functions *cost*), then their sum $\sum_{i=1}^{n} h_i(s, cost_i)$ is admissible for the original cost function *cost* due to the way the cost partitioning distributes the operator costs. The question is: how do we find a cost partitioning that achieves a high overall heuristic estimate?

Katz and Domshlak (2008, 2010) showed that optimal cost partitionings can be found in polynomial time by linear programming for a wide range of abstraction heuristics. It has been demonstrated empirically that computing optimal cost partitionings for some or even all states encountered during search is a viable approach for landmark heuristics and certain classes of implicit abstraction heuristics (Karpas & Domshlak, 2009; Katz & Domshlak, 2010; Karpas, Katz, & Markovitch, 2011). However, computing even a single optimal cost partitioning can already be prohibitively expensive for (explicit) abstractions of modest size (Pommerening, Röger, & Helmert, 2013; Seipp, Keller, & Helmert, 2017b).

Consequently, several alternatives to optimal cost partitioning with varying time vs. accuracy tradeoffs have been proposed, such as uniform cost partitioning (Katz & Domshlak, 2007) and post-hoc optimization (Pommerening et al., 2013). A drawback of these non-optimal cost partitioning algorithms is that they can assign a higher share of the costs to a heuristic than necessary, in the sense that the same heuristic values could be obtained with a lower share of the costs. Therefore, we introduce a new cost partitioning algorithm, called *saturated cost partitioning*, which does not waste costs by assigning them to heuristics that do not make use of them. Seipp, Keller, and Helmert (2017a) provide a theoretical and experimental comparison of cost partitioning algorithms, including saturated cost partitioning.

The saturated cost partitioning algorithm works as follows: given an ordered sequence of heuristics $\langle h_1, \ldots, h_n \rangle$ and an overall cost function *cost*, we compute the minimum cost function $cost_1$ with $h_1(s, cost_1) = h_1(s, cost)$ for all states s. Then we subtract $cost_1$ from *cost* and use the remaining costs $cost - cost_1$ as a new overall cost function, repeating the process for h_2, and so on. As a result, each heuristic h_i only uses the costs that it actually needs to maintain its estimates (for the given remaining costs), and the remaining costs are used to define further cost functions that allow summing the estimates of all heuristics admissibly. We use the name *saturated cost function* to refer to the minimum cost function that preserves all heuristic values. A saturated cost function is a *generalized cost function* since it may assign negative costs to operators, including $-\infty$ for operators that are only applicable in states from which no goal state can be reached.

Definition 7. Generalized cost function.
Let Π *be a planning task with operators* $\mathcal{O}$*. A* generalized cost function *cost assigns real-valued costs or negative infinity to each operator, i.e.,* $cost(o) \in \mathbb{R} \cup \{-\infty\}$ *for all* $o \in \mathcal{O}$*.*

Definition 8. Saturated cost function.
Let Π *be a planning task with operators* $\mathcal{O}$*. Let* h *be a heuristic for* Π *and cost be a (non-generalized) cost function for* $\mathcal{O}$*. The* saturated cost function $scf = saturate(h, cost)$ *is the minimum generalized cost function that preserves all heuristic estimates. Formally,*

1. $h(s, scf) = h(s, cost)$ *for all states* $s \in S(\Pi)$ *and*
2. *all other generalized cost functions* $cost'$ *with* $h(s, cost') = h(s, cost)$ *for all states* $s \in S(\Pi)$ *satisfy* $cost'(o) \geq scf(o)$ *for all operators* $o \in \mathcal{O}$*.*

In general, such a cost function does not necessarily exist because there might be no unique minimum cost function. However, for explicitly-represented abstraction heuristics h, $saturate(h, cost)$ exists and can be efficiently computed as shown in the following theorem.

Theorem 3. Saturated cost function for abstraction heuristics.
Let Π *be a planning task with operators* $\mathcal{O}$ *and cost function cost, and let* h *be an abstraction heuristic for* Π *defined by the abstract transition system* $\mathcal{T}$ *with states* S *and transitions* T*. Furthermore, let* $h^*_{\mathcal{T}}(a, cost')$ *be the cost of the cheapest abstract plan in* $\mathcal{T}$ *weighted by cost function* $cost'$ *for the abstract state* $a \in S$*. Finally, let* $T_{fin} \subseteq T$ *be the subset of transitions* $a \xrightarrow{o} b \in T$ *with* $h^*_{\mathcal{T}}(b, cost) < \infty$*. Then the saturated cost function* $saturate(h, cost)$ *is the function scf with*

$$scf(o) = \begin{cases} -\infty & \text{if } o \text{ induces no transitions in } T_{fin} \\ \max_{a \xrightarrow{o} b \in T_{fin}} (h^*_{\mathcal{T}}(a, cost) - h^*_{\mathcal{T}}(b, cost)) & \text{otherwise} \end{cases}$$

for all $o \in \mathcal{O}$*.*

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

We now show that scf satisfies the two properties of saturated cost functions from Definition 8:

Prop. 1. We must show $h(s, scf) = h(s, cost)$ for all states $s \in S(\Pi)$.

We first show $h(s, scf) \leq h(s, cost)$ for all $s \in S(\Pi)$. It is easy to see that $scf(o) \leq cost(o)$ for all operators o: in the case where $scf(o) = -\infty$, this holds trivially, and otherwise there exists a transition $a \xrightarrow{o} b$ with $scf(o) = h^*_{\mathcal{T}}(a, cost) - h^*_{\mathcal{T}}(b, cost) \leq cost(o) + h^*_{\mathcal{T}}(b, cost) - h^*_{\mathcal{T}}(b, cost) = cost(o)$, where we use that $h^*_{\mathcal{T}}(a, cost) \leq cost(o) + h^*_{\mathcal{T}}(b, cost)$ by the triangle inequality. From $scf(o) \leq cost(o)$ for all o, we get that $h(s, scf) \leq h(s, cost)$ for all states $s \in S(\Pi)$ since lowering the weights in a transition system can only decrease goal distances.

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

Let a_0 be any abstract state in S. If there is no goal trace for a_0, we have $h^*_{\mathcal{T}}(a_0, \mathit{scf}) = h^*_{\mathcal{T}}(a_0, \mathit{cost}) = \infty$. Otherwise, let $\tau = \langle a_0 \xrightarrow{o_1} a_1, \dots, a_{k-1} \xrightarrow{o_k} a_k \rangle$ be a goal trace for a_0. All transitions in τ must be part of T_{fin} because clearly the goal is reachable from all states that τ traverses. We can bound the cost of τ under *scf* by

$$\begin{aligned}
\sum_{i=1}^{k} \mathit{scf}(o_i) &\overset{(1)}{\geq} \sum_{i=1}^{k} (h^*_{\mathcal{T}}(a_{k-1}, \mathit{cost}) - h^*_{\mathcal{T}}(a_k, \mathit{cost})) \\
&\overset{(2)}{=} \sum_{i=0}^{k-1} h^*_{\mathcal{T}}(a_k, \mathit{cost}) - \sum_{i=1}^{k} h^*_{\mathcal{T}}(a_k, \mathit{cost}) \\
&\overset{(3)}{=} h^*_{\mathcal{T}}(a_0, \mathit{cost}) - h^*_{\mathcal{T}}(a_k, \mathit{cost}) \\
&\overset{(4)}{=} h^*_{\mathcal{T}}(a_0, \mathit{cost}) - 0 \\
&= h^*_{\mathcal{T}}(a_0, \mathit{cost}),
\end{aligned}$$

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

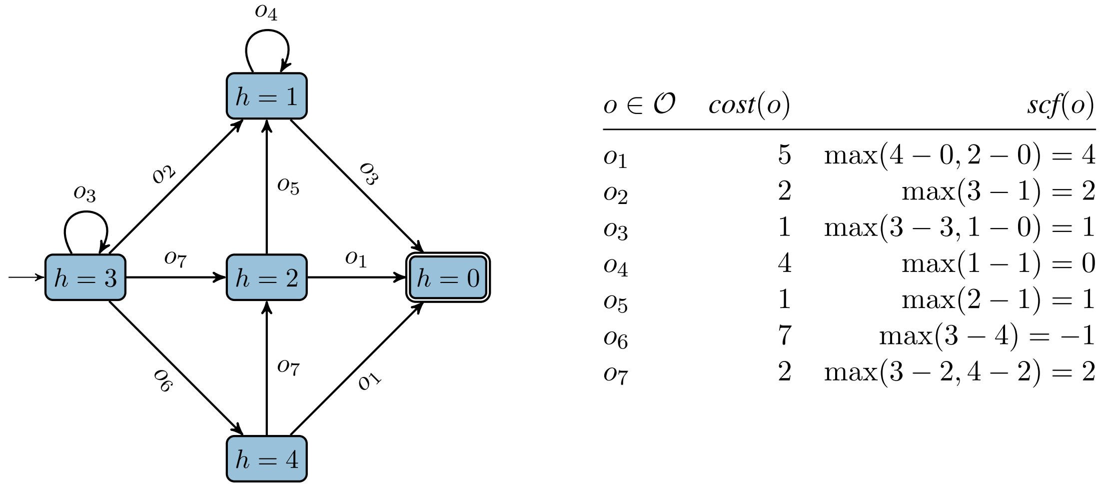
This shows that the cost of any plan for a_0 under *scf* is never lower than $h^*_{\mathcal{T}}(a_0, \mathit{cost})$, the cost of an optimal plan under *cost*. This proves $h^*_{\mathcal{T}}(a, \mathit{scf}) \geq h(a, \mathit{cost})$ for all abstract states $a \in S$, concluding this part of the proof.

Prop. 2. By contradiction: let cost' be a generalized cost function with $h(s, \mathit{cost}') = h(s, \mathit{cost})$ for all concrete states $s \in S(\Pi)$ and $\mathit{cost}'(o) < \mathit{scf}(o)$ for some operator $o \in \mathcal{O}$. Since $\mathit{cost}'(o)$ can only be lower than $\mathit{scf}(o)$ if $\mathit{scf}(o) \neq -\infty$, this means that the second case in the definition of *scf* in the statement of this theorem applies for o and we have $\mathit{cost}'(o) < \max_{a \xrightarrow{o} b \in T_{\mathit{fin}}} (h^*_{\mathcal{T}}(a, \mathit{cost}) - h^*_{\mathcal{T}}(b, \mathit{cost}))$. Therefore, there exists a transition $a \xrightarrow{o} b \in T_{\mathit{fin}}$ with $\mathit{cost}'(o) < h^*_{\mathcal{T}}(a, \mathit{cost}) - h^*_{\mathcal{T}}(b, \mathit{cost})$. This implies $h^*_{\mathcal{T}}(a, \mathit{cost}) > \mathit{cost}'(o) + h^*_{\mathcal{T}}(b, \mathit{cost})$. Because $h(s, \mathit{cost}') = h(s, \mathit{cost})$ for all concrete states $s \in S(\Pi)$, we also have $h^*_{\mathcal{T}}(c, \mathit{cost}') = h^*_{\mathcal{T}}(c, \mathit{cost})$ for all abstract states $c \in S$. With this, we obtain $h^*_{\mathcal{T}}(a, \mathit{cost}') > \mathit{cost}'(o) + h^*_{\mathcal{T}}(b, \mathit{cost}')$, which violates the triangle inequality for shortest paths in graphs.

□

Using Theorem 3 we can efficiently compute the saturated cost function for any abstraction heuristic where the abstract transition system is either explicitly represented, as for Cartesian abstractions and merge-and-shrink abstractions without label reduction (Helmert et al., 2014), or easily enumerable, as for pattern databases (Culberson & Schaeffer, 1998). For merge-and-shrink heuristics using *label reduction* (Sievers, Wehrle, & Helmert, 2014) the computation is more expensive, but still polynomial.

We demonstrate how to compute the saturated cost function for abstractions in Figure 8. It shows the abstract transition system of an example abstraction heuristic. The transitions are weighted by a



$o \in \mathcal{O}$	*cost*(o)	*scf*(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 8: Left: abstract transition system of an example planning task. Every transition is associated with an operator. Right: original costs and saturated costs that suffice to preserve all goal distances in the abstract transition system.

Algorithm 7 Saturated Cost Partitioning. For a given sequence of heuristics and a cost function, compute the saturated cost partitioning.

1: **function** SATURATEDCOSTPARTITIONING($\langle h_1, \dots, h_n \rangle$, *cost*)
2:     **for** $1 \leq i \leq n$ **do**
3:         $scf_i \leftarrow saturate(h_i, cost)$
4:         $cost \leftarrow cost - scf_i$
5:     **return** $\langle scf_1, \dots, scf_n \rangle$

given cost function *cost*. The states in the abstract transition system show the abstract goal distances ($h = X$). The table on the right in Figure 8 lists both the original and the saturated cost of each operator. Note that the transition from $h = 2$ to $h = 0$ must have at least a weight of 4. Otherwise, the transition between $h = 4$ and $h = 0$, which is induced by the same operator, would also be assigned a weight smaller than 4 and thus the goal distance for $h = 4$ would decrease.

Splitting the cost of each operator o into the cost needed for preserving the goal distances *scf*(o) and the remaining cost *cost*(o) − *scf*(o) produces a cost partitioning: we associate the saturated cost function scf_i with the current heuristic h_i and use the remaining cost $cost - scf_i$ to define further cost functions for subsequent heuristics. Algorithm 7 shows pseudo-code for the saturated cost partitioning procedure. For operators o that have a saturated cost of $-\infty$ in abstraction i, the remaining cost of o is ∞ in all subsequent abstractions. Strictly speaking, this means that the remaining cost function is no longer a normal cost function. However, such infinite costs can be handled easily by removing all transitions induced by o in these abstractions.

In our implementation, we exploit the fact that saturated cost partitioning only needs to hold one abstract transition system in memory at a time by interleaving abstraction computation and cost partitioning: we iteratively create an abstract transition system $\mathcal{T}'$ using the remaining cost function *cost*, compute the saturated cost function *scf* for $\mathcal{T}'$, subtract *scf* from *cost* and build the next abstraction using the reduced cost function.

A further modification we apply in our implementation is ignoring abstract states (and their outgoing transitions) that are unreachable from the abstract initial state. Since we want to apply the resulting heuristic to forward search, heuristic values of unreachable states are irrelevant, and using lower costs is always preferable if we can still preserve the heuristic values of all reachable states.

5.2 Multiple Abstractions of the Original Task

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

Table 1 shows the number of solved tasks from previous International Planning Competitions for $X = 900s$ and different values of n. All versions use A* and the saturated cost partitioning heuristics computed over the n abstraction heuristics to find a plan. We give each run a time limit of 1800 seconds (of which at most X seconds are used to construct the abstractions) and a memory limit of 2 GiB.

We see that building more than one abstraction is usually detrimental. In only 4 out of 40 domains it is beneficial to use multiple abstractions: we obtain the highest coverage in Logistics when using 10 abstractions, for Sokoban it is preferable to use 5 or 10 abstractions and for Openstacks and Tetris we obtain the best results with 2–50 abstractions. The total coverage score remains roughly the same when going from 1 to 10 abstractions but it decreases when we use 20 or more abstractions. We hypothesize that this is the case because the computed abstractions are too similar to each other, focusing mostly on the same parts of the task. Computing multiple abstractions does not yield a more informed additive heuristic and instead just consumes time that could have been used to further refine a single abstraction.

To see why diversifying abstractions is essential, consider the extreme case where we evaluate the same abstraction heuristic h under two different cost functions cost_1 and cost_2. For every state s we have $h(s, \mathit{cost}_1) + h(s, \mathit{cost}_2) \leq h(s, \mathit{cost}_1 + \mathit{cost}_2)$, i.e., using the sum of heuristic values is *dominated* by using h only once with cost function $\mathit{cost}_1 + \mathit{cost}_2$. (This follows from the admissibility of cost partitioning and the fact that abstraction heuristics are based on shortest paths in transition systems.) So we need to make sure that the abstractions computed in different iterations of the algorithm are sufficiently different.

There are several possible ways of ensuring such diversity within the CEGAR framework. One way is to make sure that different iterations of the CEGAR algorithm produce different results even when presented with the same input planning task. This is quite possible to do because the CEGAR algorithm has several choice points that affect its outcome, in particular in the refinement step where there are frequently multiple splits to choose from. By ensuring that these choices are resolved differently in different iterations of the algorithm, we can achieve some degree of diversification. We call this approach *diversification by refinement strategy*.

Another way of ensuring diversity, even in the case where the CEGAR algorithm always generates the same abstraction when faced with the same input task, is to modify the inputs to the CEGAR algorithm. Rather than feeding the actual planning task to the CEGAR algorithm, we can present it with different “subproblems” in every iteration, so that it will naturally generate different results. To ensure that the resulting heuristic is admissible, it is sufficient that every subproblem we use as

Abstractions	1	2	5	10	20	50	100	200	500	1000
airport	**22**	**22**	**22**	**22**	**22**	**22**	21	21	18	18
barman	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	0	0
blocks	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	16	16
depot	**5**	**5**	**5**	**5**	**5**	4	4	4	4	2
driverlog	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	8	7
elevators	**29**	**29**	**29**	**29**	**29**	**29**	**29**	26	20	16
floortile	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	0
freecell	**19**	**19**	18	18	18	17	15	15	14	14
ged	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	13	12
grid	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	1
gripper	**8**	**8**	**8**	**8**	**8**	7	7	6	6	6
hiking	**12**	**12**	**12**	**12**	**12**	**12**	**12**	11	11	10
logistics	17	17	17	**18**	17	17	17	16	13	13
miconic	**55**	**55**	**55**	**55**	**55**	**55**	**55**	52	50	45
mprime	**27**	**27**	**27**	**27**	26	**27**	25	25	24	24
mystery	**18**	**18**	17	17	17	17	17	17	17	16
nomystery	**10**	**10**	9	9	9	8	8	8	8	7
openstacks	45	**47**	**47**	**47**	**47**	**47**	42	40	31	27
parcprinter	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	14	14
pegsol	**44**	**44**	**44**	**44**	**44**	**44**	**44**	**44**	42	42
pipes-nt	**17**	**17**	**17**	**17**	**17**	**17**	16	14	13	12
pipes-t	**12**	**12**	**12**	**12**	**12**	**12**	10	10	10	7
psr-small	**49**	**49**	**49**	**49**	**49**	**49**	48	48	48	48
satellite	**6**	**6**	**6**	**6**	**6**	**6**	**6**	5	5	5
scanalyzer	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**	9	9
sokoban	39	39	**41**	**41**	39	37	32	31	27	20
storage	**15**	**15**	**15**	**15**	**15**	**15**	14	14	13	13
tetris	7	**9**	**9**	**9**	**9**	**9**	8	7	5	5
tidybot	**22**	**22**	**22**	**22**	**22**	19	19	18	12	9
tpp	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	5	5
transport	**23**	**23**	**23**	**23**	**23**	22	22	21	21	19
trucks	**8**	7	7	7	6	6	6	6	4	4
visitall	**13**	**13**	**13**	**13**	**13**	**13**	12	12	12	12
woodwork	**15**	13	13	13	13	13	9	9	9	9
zenotravel	**9**	**9**	8	8	8	8	8	8	8	8
...	...	...	...	...	...	...	...	...	...	...
Sum (1667)	682	**683**	681	682	677	668	642	624	554	515

Table 1: Number of solved tasks for a growing number of Cartesian abstractions. We omit domains in which coverage does not change and highlight best results in bold.

an input to the CEGAR algorithm is itself an abstraction of the original task. We call this approach *diversification by task modification*. We discuss these two approaches in the following sections.

5.3 Diversification by Refinement Strategy

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

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

Table 2 shows the results for this approach. We see that the h^{add}-based refinement strategy leads to better results than the default refinement strategy on average. While both methods solve the same number of tasks in the basic case of only one abstraction (682 tasks), for values of n between 2 and 1000 we obtain 16–40 additional solved tasks compared to the corresponding columns in Table 1. We also see that 10 out of 40 domains benefit from using more than one abstraction, whereas for the original refinement strategy this only holds for 4 domains. The total coverage score increases from 682 to 700–705 tasks when using 2–50 abstractions instead of a single abstraction. If we use 100 or more abstractions, total coverage decreases again.

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

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

5.4 Abstraction by Goals

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

If $v \mapsto d$ is a goal atom, we create a modified planning task which is identical to the original one except that $v \mapsto d$ is the only goal atom. This means that the original and modified task have exactly the same states and transitions and only differ in their goal states. In the original task, *all* goals need to be satisfied in a goal state, but in the modified one, *only* $v \mapsto d$ needs to be reached. The goal states of the modified task are hence a superset of the original goal states, and we can conclude that the modification defines a non-induced abstraction in the sense of Definition 3 (where the abstraction mapping is the identity function).

Abstractions	1	2	5	10	20	50	100	200	500	1000
airport	21	22	**23**	**23**	**23**	**23**	22	22	19	19
barman	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	3	0
blocks	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	**18**	16
depot	**5**	**5**	**5**	**5**	**5**	**5**	4	4	4	4
driverlog	**10**	**10**	**10**	**10**	**10**	**10**	**10**	9	9	9
elevators	**26**	**26**	**26**	**26**	**26**	**26**	**26**	20	16	14
freecell	**19**	**19**	**19**	**19**	**19**	17	17	15	14	14
ged	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	13
grid	**2**	**2**	**2**	**2**	**2**	**2**	**2**	1	1	1
gripper	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	7	6
hiking	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	10	9
logistics	18	**24**	23	23	23	22	21	20	20	19
miconic	55	**61**	60	60	60	60	58	55	52	50
mprime	**26**	**26**	**26**	**26**	**26**	25	24	24	23	23
mystery	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	16	16
nomystery	10	11	**13**	12	10	10	9	8	8	8
openstacks	**47**	**47**	**47**	**47**	**47**	**47**	43	39	36	31
parcprinter	**20**	**20**	**20**	**20**	**20**	**20**	**20**	17	18	16
pegsol	**44**	**44**	**44**	**44**	**44**	**44**	**44**	**44**	42	42
pipes-nt	**17**	**17**	**17**	**17**	**17**	**17**	**17**	16	15	13
pipes-t	**14**	**14**	**14**	**14**	**14**	**14**	12	11	10	10
psr-small	**49**	**49**	**49**	**49**	**49**	**49**	**49**	**49**	48	48
rovers	6	7	**8**	7	**8**	**8**	**8**	**8**	7	7
scanalyzer	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**	9	9
sokoban	**39**	**39**	**39**	**39**	**39**	38	37	29	24	21
storage	**15**	**15**	**15**	**15**	**15**	**15**	14	14	14	13
tetris	8	**9**	**9**	8	8	**9**	8	7	6	5
tidybot	22	24	24	24	24	**26**	24	22	18	17
tpp	6	**7**	**7**	**7**	**7**	**7**	**7**	**7**	6	6
transport	**23**	**23**	**23**	**23**	**23**	22	22	21	19	17
trucks	7	**9**	**9**	**9**	**9**	**9**	**9**	7	8	5
visitall	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	12	12
woodwork	15	15	15	15	15	**17**	**17**	13	10	9
zenotravel	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	8
…	…	…	…	…	…	…	…	…	…	…
Sum (1667)	682	703	**705**	702	701	700	682	640	588	552

Table 2: Number of solved tasks for a growing number of Cartesian abstractions. All abstractions are computed by CEGAR using the max-h^{add} refinement strategy. We omit domains in which coverage does not change and highlight best results in bold.

Algorithm 8 Construct modified task for landmark $v \mapsto d$.

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

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

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

5.5 Abstraction by Landmarks

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

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

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

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

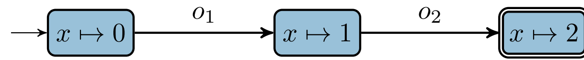


Figure 9: Example task in which operators o_1 and o_2 change the value of the single variable x from its initial value 0 to 1 and from 1 to its desired value 2.

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

Without path information, how can we decide whether a given landmark could have been reached prior to state s? The key to this question is the notion of a *possibly-before* set for atoms of delete relaxations, which has been previously considered by Porteous and Cresswell (2002). We say that an atom f' is *possibly before* atom f if f' can be achieved in the delete relaxation of the planning task without achieving f. We write $pb(f)$ for the set of atoms that are possibly before f; this set can be computed using a fixpoint computation shown in Algorithm 8 (function POSSIBLY-BEFORE). This simple implementation runs in time $O(n^2)$ for a task of size n, but a linear time version is possible by using suitable data structures. From the monotonicity properties of delete relaxations, it follows that if l is a delete-relaxation landmark and all atoms of the current state s are contained in $pb(l)$, then l still has to be achieved from s.

Based on this insight, we can construct Π_l as follows. First, we compute $pb(l)$. The modified task Π_l only contains the atoms in $pb(l)$ and l itself; all other atoms are removed. The landmark l is the only goal. The initial state and operators are identical to the original task, except that we discard operators whose preconditions are not contained in $pb(l)$ (by the definition of possibly-before sets, these can only become applicable after reaching l) and for all operators that achieve l, we make l their only effect. (Adapting such operators is necessary because they might have other effects that fall outside $pb(l)$. Note that such operators are guaranteed to achieve a goal state, and for an abstraction heuristic it does not matter which exact goal state we end up in.) The complete construction is shown in Algorithm 8.

The states $S(\Pi_l)$ of the modified task are exactly the states s of the original planning task where $s \subseteq pb(l) \cup \{l\}$. The abstraction function that is associated with the modified task maps every state in $S(\Pi_l)$ to itself. In all other states the landmark might potentially have been achieved, so they should be mapped to an arbitrary goal state of the modified task. This mapping is easy to represent within the framework of Cartesian abstraction because $S(\Pi_l)$ is a Cartesian set and its complement can be represented as the disjoint union of a small number of Cartesian sets (bounded by the number of variables of the planning task). Like in the case of abstraction by goals, the goal states of the landmark task are a superset of the goal states in the original task. Therefore, the resulting abstraction is not induced in the sense of Definition 3.

5.6 Abstraction by Landmarks: Improved

In the basic form just presented, the tasks constructed for fact landmarks do not provide as much diversification as we would desire. We illustrate the issue with the example task depicted in Figure 9. The task has three landmarks $x \mapsto 0$, $x \mapsto 1$ and $x \mapsto 2$ that must be achieved in exactly this order in every plan. When we compute the abstraction for $x \mapsto 1$, the CEGAR algorithm has to find

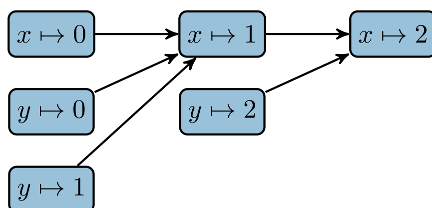


Figure 10: Example landmark orderings. Orderings implied by transitivity are omitted. For example, the ordering $x \mapsto 0 \prec x \mapsto 2$ is represented by $x \mapsto 0 \prec x \mapsto 1$ and $x \mapsto 1 \prec x \mapsto 2$. (Note that natural landmark orderings are transitive.) To avoid focusing on the same parts of the task in multiple subtasks, the improved abstraction-by-landmarks procedure combines the atoms $y \mapsto 0$ and $y \mapsto 1$ before building the abstraction for $x \mapsto 1$. For the subtask $x \mapsto 2$, it combines the atoms $y \mapsto 0$, $y \mapsto 1$, $y \mapsto 2$, and it also combines the atoms $x \mapsto 0$ and $x \mapsto 1$.

a plan for getting from $x \mapsto 0$ to $x \mapsto 1$. Similarly, the abstraction procedure for $x \mapsto 2$ has to return a plan that takes us from $x \mapsto 0$ to $x \mapsto 2$. Since going from $x \mapsto 0$ to $x \mapsto 2$ includes the subproblem of going from $x \mapsto 0$ to $x \mapsto 1$, we have to find a plan from $x \mapsto 0$ to $x \mapsto 1$ twice, which runs counter to our objective of finding abstractions that focus on different aspects on the planning task.

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

In detail, the alternative construction proceeds as follows. We start by performing the basic landmark task construction described in Algorithm 8, resulting in a planning task for landmark l which we denote by Π_l.

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

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

For example, consider again the landmark $l = x \mapsto 2$ in the example task from Figure 9. We detect that $x \mapsto 0$ and $x \mapsto 1$ are landmarks that must be achieved before l. They both refer to the variable x, so we combine the values 0 and 1 into a single value, i.e., we partition $dom(x)$ into $\{0, 1\}$ and $\{2\}$ in the domain abstraction. The effect of this is that in the task for l, we no longer need to find a subplan from $x \mapsto 0$ to $x \mapsto 1$. Figure 10 illustrates the procedure with a slightly more complex example.

6. Experiments

We implemented our counterexample-guided Cartesian abstraction refinement and saturated cost partitioning algorithms in the Fast Downward planning system (Helmert, 2006). For running exper-

iments, we use the Downward Lab toolkit (Seipp, Pommerening, Sievers, & Helmert, 2017c). Our benchmark set consists of all 1667 tasks from the optimization tracks of the 1998–2014 International Planning Competitions. All tasks are given in the PDDL format (Fox & Long, 2003), and we use the translator component of Fast Downward to convert them into SAS^+ tasks (Helmert, 2009). In our analyses we ignore the time taken for this conversion since it is the same for all compared algorithms. Our code, benchmarks and experimental data are available online.[1]

We evaluate five ways of creating Cartesian abstraction heuristics with counterexample-guided abstraction refinement: a single abstraction (h^{CEGAR}, Section 4), abstraction by goals ($h^{\text{CEGAR}}_{s_\star}$, Section 5.4), abstraction by landmarks ($h^{\text{CEGAR}}_{\text{LM}}$, Section 5.5), improved abstraction by landmarks ($h^{\text{CEGAR}}_{\text{LM+}}$, Section 5.6), and a combination of the latter two methods ($h^{\text{CEGAR}}_{\text{LM+}s_\star}$), which we describe below.

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

- h^{iPDB}: the canonical heuristic using pattern databases found by 15 minutes of hill climbing (Haslum, Botea, Helmert, Bonet, & Koenig, 2007; Sievers, Ortlieb, & Helmert, 2012)
- $h^{\text{M\&S}}$: merge-and-shrink using bisimulation, the DFP merge strategy and at most 100 000 abstract states (Helmert et al., 2014; Sievers et al., 2014)
- h^{PhO}: post-hoc optimization using systematic patterns of sizes 1 and 2 (Pommerening et al., 2013)

We apply a time limit of 30 minutes and a memory limit of 2 GiB to all algorithm runs and let all versions that use CEGAR refine for at most 15 minutes. We also stop refining and start the A^* search when the refinement loop is about to exhaust the 2 GiB memory limit. For the CEGAR versions using multiple (additive) abstraction heuristics we distribute the refinement time equally among the abstractions. We let the saturated cost partitioning algorithm consider the subtasks in a randomized order.[2] Table 3 shows the number of solved instances from the benchmark set for the compared heuristics.

6.1 Comparison of CEGAR to Other Abstraction Heuristics

We begin our analysis by comparing the basic h^{CEGAR} heuristic based on a single abstraction to the heuristics from the literature. While the total coverage of h^{CEGAR} (682 tasks) is lower than that of h^{iPDB} (860 tasks), $h^{\text{M\&S}}$ (752 tasks) and h^{PhO} (737 tasks), it outperforms them in several domains. Table 4 compares the heuristics on a per-domain basis and shows that h^{CEGAR} solves more tasks than h^{iPDB}, $h^{\text{M\&S}}$ and h^{PhO} in 2, 7 and 13 domains, respectively. However, the opposite is true in 27, 25 and 15 domains.

Although h^{CEGAR} is outperformed by h^{iPDB}, the former still has a theoretical advantage over the latter. While the CEGAR loop is guaranteed to converge to a plan, h^{iPDB} can get stuck in

1. Code: `https://doi.org/10.5281/zenodo.1240992`
Benchmarks: `https://doi.org/10.5281/zenodo.1205019`
Experimental data: `https://doi.org/10.5281/zenodo.1240998`
2. To evaluate the influence of randomness, we ran the $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ version ten times with different random seeds in a preliminary experiment. The resulting arithmetic mean of the total coverage scores was 782.9 with a sample standard deviation of 4.09 tasks. Due to this low variance, we only use a single random seed for each randomized algorithm.

	h^{iPDB}	$h^{\text{M\&S}}$	h^{PhO}	h^{CEGAR}	$h^{\text{CEGAR}}_{s_\star}$	$h^{\text{CEGAR}}_{\text{LM}}$	$h^{\text{CEGAR}}_{\text{LM+}}$	$h^{\text{CEGAR}}_{\text{LM+}s_\star}$
airport (50)	29	18	27	22	31	30	29	**33**
barman (34)	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
blocks (35)	**28**	22	26	18	18	18	18	18
childsnack (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
depot (22)	**11**	6	7	5	6	5	6	6
driverlog (20)	**13**	**13**	**13**	10	**13**	**13**	**13**	**13**
elevators (50)	**40**	29	36	29	37	37	37	37
floortile (40)	2	**8**	2	2	2	2	2	2
freecell (80)	20	20	15	19	17	17	**34**	33
ged (20)	**19**	17	15	15	15	15	15	15
grid (5)	**3**	2	2	2	2	2	2	2
gripper (20)	8	**20**	7	8	8	8	8	8
hiking (20)	12	**13**	11	12	12	11	12	12
logistics (63)	**30**	25	26	17	26	26	22	27
miconic (150)	67	**74**	54	55	62	65	70	71
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	23	24	21	**27**	**27**	**27**	**27**	**27**
mystery (30)	16	16	15	**18**	**18**	**18**	**18**	**18**
nomystery (20)	**20**	18	16	10	14	14	14	14
openstacks (100)	**47**	**47**	**47**	45	**47**	**47**	**47**	**47**
parcprinter (50)	**39**	24	30	18	22	20	34	34
parking (40)	**13**	1	3	0	0	0	0	0
pathways (30)	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
pegsol (50)	**48**	**48**	44	44	44	44	44	44
pipes-nt (50)	**21**	16	15	17	17	17	17	17
pipes-t (50)	**17**	15	9	12	11	14	14	14
psr-small (50)	49	**50**	49	49	49	49	49	49
rovers (40)	**8**	**8**	7	6	7	7	7	7
satellite (36)	6	**7**	6	6	6	6	6	6
scanalyzer (50)	**23**	**23**	11	21	21	21	21	21
sokoban (50)	**50**	48	49	39	41	40	41	41
storage (30)	**16**	15	15	15	**16**	**16**	**16**	**16**
tetris (17)	**10**	2	3	7	9	9	9	9
tidybot (40)	22	1	21	22	24	**26**	24	25
tpp (30)	6	7	6	6	**11**	8	6	7
transport (70)	**33**	25	21	23	23	23	23	23
trucks (30)	9	7	7	8	10	**12**	**12**	**12**
visitall (40)	**28**	13	27	13	13	13	13	13
woodwork (50)	23	20	**25**	15	15	19	19	19
zenotravel (20)	**13**	12	11	9	12	12	12	12
Sum (1667)	**860**	752	737	682	744	749	779	790

Table 3: Coverage scores by domain for different heuristics. We highlight best results in bold.

	h^{iPDB}	$h^{\text{M\&S}}$	h^{PhO}	h^{CEGAR}	$h^{\text{CEGAR}}_{s_\star}$	$h^{\text{CEGAR}}_{\text{LM}}$	$h^{\text{CEGAR}}_{\text{LM+}}$	$h^{\text{CEGAR}}_{\text{LM+}s_\star}$	Coverage
h^{iPDB}	–	**21**	**29**	**27**	**22**	**23**	**20**	**20**	**860**
$h^{\text{M\&S}}$	8	–	**19**	**25**	**19**	**20**	**19**	**17**	752
h^{PhO}	1	12	–	**15**	8	8	8	7	737
h^{CEGAR}	2	7	13	–	2	2	0	0	682
$h^{\text{CEGAR}}_{s_\star}$	6	11	**18**	**17**	–	**6**	3	1	744
$h^{\text{CEGAR}}_{\text{LM}}$	6	11	**17**	**18**	5	–	4	2	749
$h^{\text{CEGAR}}_{\text{LM+}}$	6	11	**18**	**19**	**6**	**6**	–	1	779
$h^{\text{CEGAR}}_{\text{LM+}s_\star}$	8	12	**20**	**20**	**9**	**8**	**5**	–	790

Table 4: Left: Pairwise comparison of different abstraction heuristics. The entry in row r and column c holds the number of domains in which heuristic r solved more tasks than heuristic c. For each heuristic pair we highlight the maximum of the entries $\langle r, c\rangle$ and $\langle c, r\rangle$ in bold. Right: Total number of solved tasks by each heuristic.

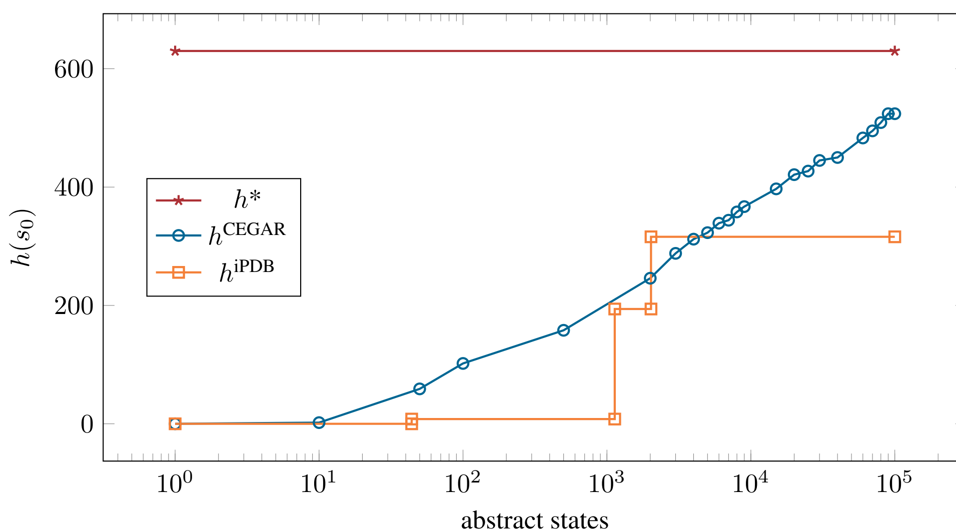


Figure 11: Initial state heuristic values for transport-08 #23.

local minima. Also, the h^{CEGAR} estimates tend to grow much more smoothly towards the perfect estimates than the ones by h^{iPDB}. Figure 11 illustrates this by comparing how the cost estimates for the initial state grow with the number of abstract states on an example task. The h^{CEGAR} estimates are generally higher than those of h^{iPDB}. This behavior can be observed in many domains.

6.2 Comparison of CEGAR Heuristics

Comparing the results for h^{CEGAR} and $h^{\text{CEGAR}}_{s_\star}$, we see that decomposing the task by goals and finding multiple abstractions separately instead of using only a single abstraction raises the number of solved tasks from 682 to 744. This substantial improvement is due to the fact that 17 domains profit from using $h^{\text{CEGAR}}_{s_\star}$ while coverage decreases in only 2 domains (see Table 4).

In total, $h^{\text{CEGAR}}_{s_\star}$ and $h^{\text{CEGAR}}_{\text{LM}}$ solve roughly the same number of tasks (744 and 749), and also for the individual domains coverage does not change much between the two heuristics: $h^{\text{CEGAR}}_{s_\star}$ solves more tasks than $h^{\text{CEGAR}}_{\text{LM}}$ in 6 domains, while the opposite is true in 5 domains. Only when we employ the improved $h^{\text{CEGAR}}_{\text{LM+}}$ heuristic that uses domain abstraction to avoid duplicate work during the refinement process, the number of solved tasks increases to 779.

6.3 Abstraction by Landmarks and Goals

In Table 4, we can observe that $h^{\text{CEGAR}}_{s_\star}$ and $h^{\text{CEGAR}}_{\text{LM+}}$ outperform each other in multiple domains: for maximum coverage $h^{\text{CEGAR}}_{s_\star}$ is preferable in 3 domains, whereas $h^{\text{CEGAR}}_{\text{LM+}}$ should be preferred in 6 domains. This suggests trying to combine the two approaches.

We do so by first computing abstractions for all subproblems returned by the *abstraction by landmarks* method. If afterwards the refinement time has not been consumed, we also calculate abstractions for the subproblems returned by the *abstraction by goals* decomposition strategy for the remaining time. The results for this approach ($h^{\text{CEGAR}}_{\text{LM+}s_\star}$) are shown in the last column in Table 3. Not only does this approach solve as many problems as the better performing ingredient technique in many individual domains, but it sometimes even outperforms both original diversification methods, raising the total number of solved tasks to 790.

6.4 Single vs. Multiple Abstractions

Comparing $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ to h^{CEGAR}, we see that $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ solves 108 more tasks than h^{CEGAR} (790 vs. 682). This difference in coverage of 15.8% is substantial because in most domains solving an additional task optimally becomes exponentially more difficult as the tasks get larger.

The big increase in coverage can be explained by the fact that for the majority of tasks $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ estimates the plan cost much better than h^{CEGAR}, as shown in Figure 12. One might expect that the increased informativeness would come with a time penalty, but in Figure 13 we can see that in fact $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ takes *less* time to compute the abstractions than h^{CEGAR}. Since all individual CEGAR invocations only stop if they run out of time or find a concrete plan, Figure 13 tells us that for most tasks h^{CEGAR} does not find a plan, but instead uses the full 15 minutes for the refinement whereas $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ almost always needs less time.

While h^{CEGAR} has a lower total coverage than the three abstraction heuristics from the literature, $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ outperforms $h^{\text{M\&S}}$ and h^{PhO} with regard to total coverage. In Table 4 we see that $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ solves more tasks than h^{iPDB}, $h^{\text{M\&S}}$ and h^{PhO} in 8, 12 and 20 domains, respectively. The opposite is true in 20, 17 and 7 domains.

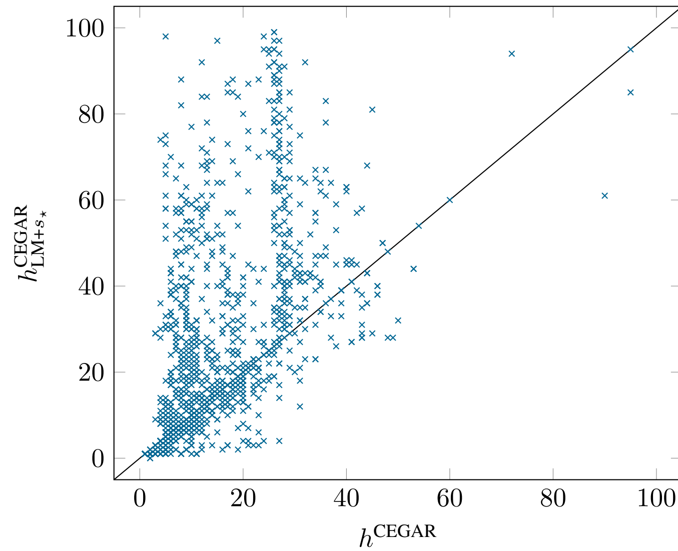


Figure 12: Comparison of the heuristic estimates for the initial state made by h^{CEGAR} and $h^{\text{CEGAR}}_{\text{LM+}s_\star}$. For better visibility, we only include tasks for which both heuristics estimate the plan cost to be at most 100. Points above the diagonal represent tasks for which $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ yields a better estimate than h^{CEGAR}.

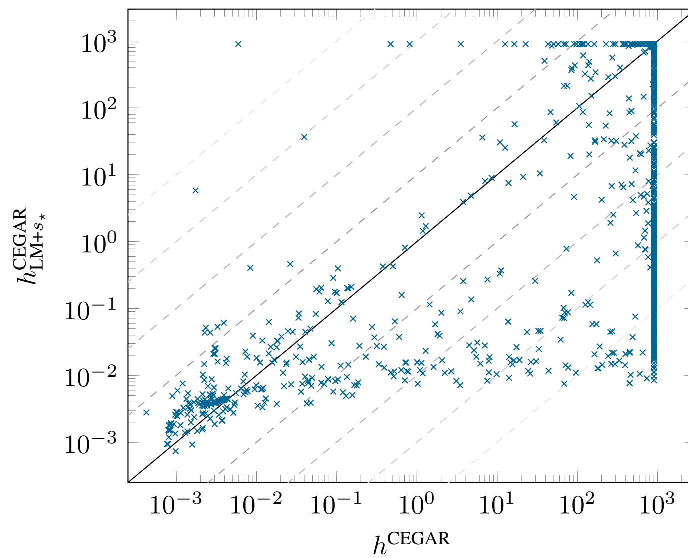


Figure 13: Comparison of the time in seconds taken by h^{CEGAR} and $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ to compute all abstractions on the tasks from Table 3. Points below the diagonal represent tasks for which $h^{\text{CEGAR}}_{\text{LM+}s_\star}$ needs less time for the computation than h^{CEGAR}.

6.5 Heuristic Orders

The strongest heuristic in this comparison, h^{iPDB}, uses a collection of pattern database heuristics. In each state h^{iPDB} computes the canonical heuristic, i.e., the maximum over all sums of heuristic values of maximal independent heuristic subsets. Seipp et al. (2017b) used a similar approach for saturated cost partitioning. They showed that maximizing over saturated cost partitioning heuristics computed for multiple different orders increases heuristic accuracy significantly compared to a single order. As a result, computing the maximum over multiple saturated cost partitioning heuristics using Cartesian abstractions of landmark and goal task decompositions leads to more solved tasks than iPDB (Seipp et al., 2017b). We can obtain even stronger results by adding pattern database heuristics to the set of component heuristics (Seipp, 2017).

7. Related Work

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

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

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

Smaus and Hoffmann (2009) have explored the idea of using CEGAR to derive informative heuristics for model checking, although not with a focus on optimality. They use CEGAR to iteratively refine predicate abstraction heuristics for directed model-checking of timed automata via greedy best-first search. As in our evaluation, they show that computing separate abstractions for each error condition is often beneficial. Since they do not enforce admissibility, they can combine the heuristics by maximizing or summing over them. Their results also show that there is no advantage in refining multiple paths instead of a single path in each iteration of the refinement loop. It remains to be tested if the same holds in our setting.

Despite the similarity between model checking and planning, CEGAR has not been thoroughly explored by the planning community. The work that comes closest to ours in a planning setting uses CEGAR for stochastic perfect information games, a generalization of Markov decision processes (Chatterjee, Henzinger, Jhala, & Majumdar, 2005). Stochastic perfect information games model two adversarial players and an uncertain environment. Markov decision processes and deterministic single-agent search problems are covered as special cases by the model. (Consider first the special

case where player 2 never gets to act, and then the further special case where no random choices occur.)

The authors propose an algorithm that iteratively refines an abstraction of the game using CEGAR. In each step, the algorithm searches for a winning strategy for player 1 in the abstract game. If it can find such a winning strategy, then player 1 also wins in the concrete game because the abstraction underapproximates player 1 and overapproximates player 2. If player 2 wins in the abstract game, the algorithm checks if the proposed abstract winning strategy can be turned into a concrete winning strategy. If yes, player 2 wins in the concrete game. If not, the algorithm attempts to refine the abstraction and continues the CEGAR loop.

Unfortunately, the paper has several critical technical errors which make the main contribution (Algorithms 1 and 2) unsound. One issue is that the formalization requires that every state has an outgoing transition, but the considered notion of abstraction fails to preserve this requirement due to its use of underapproximation for player-1 states. If we restrict attention to transition systems where this issue does not arise, the algorithm is still incorrect because the proposed refinement operators *Focus* and *ValueFocus* are insufficient to identify all relevant refinements.[3]

The authors also propose a variant of their algorithm for deterministic transition systems. It first compiles the planning task to a Boolean formula and then iteratively refines an abstraction that takes more and more Boolean variables into account. The paper contains no experimental evaluation or indication that either algorithm variant has been implemented. Both variants are based on blind search, and we believe they are very unlikely to deliver competitive performance.

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

Another difference to our work is that Haslum (2012) uses his algorithm to prove lower bounds, whereas we use the obtained abstractions to compute admissible heuristics for A* search. In principle, both methods can be used for both purposes, but Haslum's algorithm suffers from the fact that the number of operators grows exponentially in the number of added conjunctions. This makes evaluating a heuristic based on Haslum's algorithm in a large number of states infeasible (Keyder, Hoffmann, & Haslum, 2012).

Keyder et al. (2012) address this problem by proposing a new refinement scheme for delete relaxations which uses conditional effects to keep the number of operators linear in the number of added conjunctions. They iteratively refine the relaxed task until a given resource limit is hit.

3. For example, consider a transition system (no random states, no player-2 states) with the states $\{I, A, B, C\}$, initial state I, reward 2 for I, reward 1 for A and B, reward 0 for C, and the following transitions: $I \to A$, $A \to A$, $A \to C$, $B \to C$, and $C \to B$. The maximal achievable average reward is 1, by going from I to A, then looping in A forever. Suppose that the goal is to achieve an average reward of at least 0.75. Starting from the initial abstraction which separates $\{I\}$ from $\{A, B, C\}$, the proposed algorithm would split $\{A, B, C\}$ into $\{A, B\}$ and $\{C\}$ using the *ValueFocus* operator, compute a maximal achievable average reward of 0.5 for the resulting abstract transition system (note that this transition system contains no self-loop in $\{A, B\}$ due to the use of underapproximation, as B has no outgoing transitions to A or B), and then falsely report that no larger value can be obtained.

Afterwards, they compute a satisficing solution for the relaxed task with the FF heuristic (Hoffmann & Nebel, 2001) and use it as a heuristic in greedy best-first search.

The same authors showed later that both Haslum's original compilation method (Haslum, 2012) and their compilation method using conditional effects (Keyder et al., 2012) greatly benefit from ignoring operators that violate *mutexes* (Keyder, Hoffmann, & Haslum, 2014). In fact, their evaluation shows that Haslum's approach together with mutex pruning is the compilation method of choice when refining relaxed tasks for LM-Cut (Helmert & Domshlak, 2009).

Fickert and Hoffmann (2017) use Keyder et al.'s refinement algorithm to refine relaxed tasks online during search. Whenever they encounter local minima of the resulting heuristic, they learn new atom conjunctions and use them to improve the heuristic. Seipp (2012) briefly touches on online Cartesian abstraction refinement with CEGAR at the end of his master's thesis, but the preliminary results are rather discouraging. The topic is studied in significantly more depth by Eifler and Fickert (2018). They refine a set of Cartesian abstractions (obtained with the goal decomposition method) online during search and maximize over multiple saturated cost partitionings computed for different orders of the abstractions. When limiting the time spent for online refinement, their approach solves more tasks in many domains than a heuristic based on a single Cartesian abstraction refined offline. However, it is not clear whether this improvement is due to the online refinement or due to their use of multiple orders.

8. Conclusion

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

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

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

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

Acknowledgments

This work was supported by the European Research Council as part of the project "State Space Exploration: Principles, Algorithms and Applications".

Appendix A. Pseudo-Code for Efficiently Rewiring Transitions

Algorithm 9 Rewiring of incoming transitions. The refinement in progress splits state $[s]$ into states d and e on variable v. This means that d and e partition $[s]$ and for all variables u, $\mathit{dom}(u, v) = \mathit{dom}(u, e)$ iff $u \neq v$. For the old transition $a \xrightarrow{o} [s]$ this procedure adds new transitions from a to the new states d and e where necessary.

```
1: procedure REWIREINCOMINGTRANSITION(a -o-> [s], d, e, v)
2:   if v ∉ vars(post(o)) then
3:     if dom(v, a) ∩ dom(v, d) ≠ ∅ then
4:       ADDTRANSITION(a, o, d)
5:     if dom(v, a) ∩ dom(v, e) ≠ ∅ then
6:       ADDTRANSITION(a, o, e)
7:   else if post(o)[v] ∈ dom(v, d) then
8:     ADDTRANSITION(a, o, d)
9:   else
10:    ADDTRANSITION(a, o, e)
```

Algorithm 10 Rewiring of outgoing transitions. The refinement in progress splits state $[s]$ into states d and e on variable v. This means that d and e partition $[s]$ and for all variables u, $\mathit{dom}(u, v) = \mathit{dom}(u, e)$ iff $u \neq v$. For the old transition $[s] \xrightarrow{o} b$ this procedure adds new transitions from the new states d and e to b where necessary.

```
1: procedure REWIREOUTGOINGTRANSITION([s] -o-> b, d, e, v)
2:   if v ∉ vars(post(o)) then
3:     if dom(v, d) ∩ dom(v, b) ≠ ∅ then
4:       ADDTRANSITION(d, o, b)
5:     if dom(v, e) ∩ dom(v, b) ≠ ∅ then
6:       ADDTRANSITION(e, o, b)
7:   else if v ∉ vars(pre(o)) then
8:     ADDTRANSITION(d, o, b)
9:     ADDTRANSITION(e, o, b)
10:  else if pre(o)[v] ∈ dom(v, d) then
11:    ADDTRANSITION(d, o, b)
12:  else
13:    ADDTRANSITION(e, o, b)
```

Algorithm 11 Rewiring of self-loops. The refinement in progress splits state $[s]$ into states d and e on variable v. This means that d and e partition $[s]$ and for all variables u, $dom(u, v) = dom(u, e)$ iff $u \neq v$. This procedure adds new self-loops and/or transitions between the new states d and e for the old self-loop $[s] \xrightarrow{o} [s]$ where necessary.

```
1: procedure REWIRESELFLOOP([s] -o-> [s], d, e, v)
2:   if v ∉ vars(pre(o)) then
3:     if v ∉ vars(post(o)) then
4:       ADDSELFLOOP(d, o)
5:       ADDSELFLOOP(e, o)
6:     else if post(o)[v] ∈ dom(v, d) then
7:       ADDSELFLOOP(d, o)
8:       ADDTRANSITION(e, o, d)
9:     else
10:      ADDTRANSITION(d, o, e)
11:      ADDSELFLOOP(e, o)
12:  else if pre(o)[v] ∈ dom(v, d) then
13:    if post(o)[v] ∈ dom(v, d) then
14:      ADDSELFLOOP(d, o)
15:    else
16:      ADDTRANSITION(d, o, e)
17:  else
18:    if post(o)[v] ∈ dom(v, d) then
19:      ADDTRANSITION(e, o, d)
20:    else
21:      ADDSELFLOOP(e, o)
```

References

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS$^+$ planning. *Computational Intelligence*, *11*(4), 625–655.

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.

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

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

Chatterjee, K., Henzinger, T. A., Jhala, R., & Majumdar, R. (2005). Counterexample-guided planning. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI 2005)*, pp. 104–111.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, *50*(5), 752–794.

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model Checking*. The MIT Press.

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

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.

Eifler, R., & Fickert, M. (2018). Online refinement of Cartesian abstraction heuristics. In *ICAPS 2018 Workshop on Heuristics and Search for Domain-independent Planning*.

Fickert, M., & Hoffmann, J. (2017). Complete local search: Boosting hill-climbing through online relaxation refinement. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pp. 107–115. AAAI Press.

Fox, M., & Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, *20*, 61–124.

Haslum, P. (2012). Incremental lower bounds for additive cost planning problems. In McCluskey, L., Williams, B., Silva, J. R., & Bonet, B. (Eds.), *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, pp. 74–82. AAAI Press.

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. (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, *173*, 503–535.

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.

Helmert, M., Röger, G., & Sievers, S. (2015). On the expressive power of non-linear merge-and-shrink representations. 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. 106–114. AAAI Press.

Hernádvölgyi, I. T., & Holte, R. C. (2000). Experiments with automatically created memory-based heuristics. In Choueiry, B. Y., & Walsh, T. (Eds.), *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA 2000)*, Vol. 1864 of *Lecture Notes in Artificial Intelligence*, pp. 281–290. Springer-Verlag.

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

Hoffmann, J., Porteous, J., & Sebastia, L. (2004). Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, *22*, 215–278.

Holte, R. C. (2013). Korf's conjecture and the future of abstraction-based heuristics. In Frisch, A. M., & Gregory, P. (Eds.), *Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation (SARA 2013)*, pp. 128–131. AAAI Press.

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.

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.

Keyder, E., Hoffmann, J., & Haslum, P. (2012). Semi-relaxed plan heuristics. In McCluskey, L., Williams, B., Silva, J. R., & Bonet, B. (Eds.), *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, pp. 128–136. AAAI Press.

Keyder, E., Hoffmann, J., & Haslum, P. (2014). Improving delete relaxation heuristics through explicitly represented conjunctions. *Journal of Artificial Intelligence Research*, *50*, 487–533.

Keyder, E., Richter, S., & Helmert, M. (2010). Sound and complete landmarks for and/or graphs. In Coelho, H., Studer, R., & Wooldridge, M. (Eds.), *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pp. 335–340. IOS Press.

Korf, R. E. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, pp. 700–705. AAAI Press.

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

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.

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

Richter, S., Helmert, M., & Westphal, M. (2008). Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pp. 975–982. AAAI Press.

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

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., Keller, T., & Helmert, M. (2017a). A comparison of cost partitioning algorithms for optimal classical planning. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pp. 259–268. AAAI Press.

Seipp, J., Keller, T., & Helmert, M. (2017b). Narrowing the gap between saturated and optimal cost partitioning for classical planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*, pp. 3651–3657. AAAI Press.

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

Sievers, S. (2017). *Merge-and-shrink Abstractions for Classical Planning: Theory, Strategies, and Implementation*. Ph.D. thesis, University of Basel.

Sievers, S., Ortlieb, M., & Helmert, M. (2012). Efficient implementation of pattern database heuristics for classical planning. In Borrajo, D., Felner, A., Korf, R., Likhachev, M., Linares López, C., Ruml, W., & Sturtevant, N. (Eds.), *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, pp. 105–111. AAAI Press.

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.

Smaus, J.-G., & Hoffmann, J. (2009). Relaxation refinement: A new method to generate heuristic functions. In Peled, D. A., & Wooldridge, M. J. (Eds.), *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MoChArt 2008)*, pp. 147–165.