Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)

Computing Domain Abstractions for Optimal Classical Planning with Counterexample-Guided Abstraction Refinement

Raphael Kreft, Clemens Büchner, Silvan Sievers, Malte Helmert

University of Basel, Switzerland
{r.kreft,clemens.buechner,silvan.sievers,malte.helmert}@unibas.ch

Abstract

Abstraction heuristics are the state of the art in optimal classical planning as heuristic search. A popular method for computing abstractions is the counterexample-guided abstraction refinement (CEGAR) principle, which has successfully been used for projections, which are the abstractions underlying pattern databases, and Cartesian abstractions. While projections are simple and fast to compute, Cartesian abstractions subsume projections and hence allow more fine-grained abstractions, however at the expense of efficiency. Domain abstractions are a third class of abstractions between projections and Cartesian abstractions in terms of generality. Yet, to the best of our knowledge, they are only briefly considered in the planning literature but have not been used for computing heuristics yet. We aim to close this gap and compute domain abstractions by using the CEGAR principle. Our empirical results show that domain abstractions compare favorably against projections and Cartesian abstractions.

Introduction

Classical planning (Ghallab, Nau, and Traverso 2004) is the problem of finding a sequence of deterministic actions leading from a specified initial world state to a desired goal configuration. A popular approach for optimally solving classical planning tasks is the A* algorithm (Hart, Nilsson, and Raphael 1968) with *admissible heuristics* (Pearl 1984). A state-of-the-art class of admissible heuristics is based on *abstractions*. There is a hierarchy of three popular classes of abstractions: *projections* are the abstractions underlying *pattern databases* (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001), *Cartesian abstractions* (Seipp and Helmert 2018) generalize projections, and *merge-and-shrink* abstractions (e.g., Helmert et al. 2014; Sievers and Helmert 2021) are the most general abstractions.

Recently, the *counterexample-guided abstraction refinement* (CEGAR) principle, which originates from model checking (Clarke et al. 2000), has successfully been used in planning for computing projections (Rovner, Sievers, and Helmert 2019a) and *Cartesian abstractions* (Seipp and Helmert 2018). CEGAR starts with a coarse abstraction of the given task and iteratively refines it based on failures of applying abstract solutions to the original task. For the refinement step, CEGAR needs to be able to efficiently retrieve the pre-image of abstract states. This is the case for the simple and fast-to-compute projections. However, they do not allow fine-grained refinement, which is why Seipp and Helmert (2018) introduce Cartesian abstractions, which are more fine-grained but also more expensive to compute. Merge-and-shrink abstractions do not allow efficient retrieval of pre-images of abstract states.

There is a fourth class of abstractions, namely *domain abstractions* (Hernádvölgyi and Holte 2000). Seipp and Helmert (2018) briefly discuss them when introducing Cartesian abstractions, but to the best of our knowledge, they have not been used for computing heuristics in planning yet. In terms of generality, they are between projections and Cartesian abstractions: they allow abstracting *some* values of a domain of a variable, thus subsuming projections (when abstracting all values), but do so globally which is more restrictive than Cartesian abstractions in general. Refining domain abstractions is therefore very efficient.

In this paper, we investigate how domain abstractions compare against PDBs and Cartesian abstractions. To this end, we adapt the CEGAR algorithm for computing patterns to the computation of domain abstractions. We present several strategies for refinement and for obtaining collections of diverse domain abstractions. We use *saturated cost partitioning* (Seipp, Keller, and Helmert 2020) to combine collections of abstraction heuristics. Our empirical evaluation shows that domain abstractions solve more tasks than PDBs and Cartesian abstractions in isolation.

Background

Classical Planning We consider classical planning in the SAS$^+$ formalism (Bäckström and Nebel 1995). A *planning task* is a 4-tuple $\Pi = \langle V, s_0, G, A \rangle$. The *state variables* $V = \langle v_1, \ldots, v_n \rangle$ are associated with finite *domains* $D_1, \ldots, D_n$. We abuse notation by treating V as sets occasionally. A *partial state* s is a set of *atoms* $v_i \mapsto d_i$ over different variables $v_i \in \mathit{Vars}(s)$ such that $d_i \in D_i$. If $\mathit{Vars}(s) = V$, s is called a *state* and we write $s = \langle d_1, \ldots, d_n \rangle$. The state s_0 is called *initial state* and G is a partial state called *goal*. Finally, A is a finite set of *actions* where each action $a \in A$ is associated with two partial states $\mathit{pre}(a)$ and $\mathit{eff}(a)$ called *precondition* and *effect*, respectively. Furthermore, each action has a

non-negative *cost* $cost(a) \in \mathbb{R}_0^+$. An action a is *applicable* in a state s iff $pre(a) \subseteq s$. The application of a in s leads to the *successor state* $s[\![a]\!] = \{v \mapsto d \in \mathit{eff}(a) \mid v \in \mathit{Vars}(\mathit{eff}(a))\} \cup \{v \mapsto d \in s \mid v \notin \mathit{Vars}(\mathit{eff}(a))\}$.

A planning task $\Pi = \langle V, s_0, G, A \rangle$ induces the transition system $\mathcal{T}^\Pi = \langle S, A, T, s_0, S_* \rangle$ where S is the set of states of Π, $T \subseteq S \times A \times S$ such that $\langle s, a, t \rangle \in T$ iff a is applicable in s and $t = s[\![a]\!]$, and $S_* = \{s \in S \mid G \subseteq s\}$. An s-plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a path in $\mathcal{T}$ leading from s to some goal state in S_*. Its cost is $\sum_{i=1}^{n} cost(a_i)$. We say π is *optimal* if it has minimal cost among all s-plans. A plan for Π is an s_0-plan. Optimal planning is the problem of finding an optimal plan or showing that no plan exists.

Abstraction Heuristics We use the A* search algorithm (Hart, Nilsson, and Raphael 1968) with an *admissible heuristic* to find optimal plans. A heuristic $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ maps every state s to an estimation of the *perfect heuristic* $h^*(s)$, which denotes the cost of an optimal s-plan. A heuristic h is admissible iff $h(s) \leq h^*(s)$ for all $s \in S$.

Let $\mathcal{T} = \langle S, A, T, s_0, S_* \rangle$ be a transition system induced by a planning task Π. An *abstraction* for $\mathcal{T}$ (and with that, for Π) is a function $\alpha : S \to S^\alpha$ where S^α is a set of *abstract states*. It induces the abstract transition system $\mathcal{T}^\alpha = \langle S^\alpha, A, T^\alpha, \alpha(s_0), S_*^\alpha \rangle$ where $T^\alpha = \{\langle \alpha(s), a, \alpha(t) \rangle \mid \langle s, a, t \rangle \in T\}$, and $S_*^\alpha = \{\alpha(s) \mid s \in S_*\}$. The *abstraction heuristic* for $\mathcal{T}$ induced by α is defined as $h^\alpha(s) = h^*(\alpha(s))$, i.e, the perfect heuristic of the abstract state in $\mathcal{T}^\alpha$.

We use the notation by Seipp and Helmert (2018) in the following. Let $V = \langle v_1, \ldots, v_n \rangle$. A set of states over V is *Cartesian* if it can be written as $A_1 \times \cdots \times A_n$ with $A_i \subseteq D_i$ for $1 \leq i \leq n$. An abstraction α is Cartesian if all its abstract states are Cartesian sets.[1]

A *projection* is an abstraction based on a subset $P \subseteq V$ of the variables called the *pattern*. It maps states s to the partial state $s|_P = \{v \mapsto d \in s \mid v \in P\}$. Projections are a special case of Cartesian abstractions: for state $s = \langle d_1, \ldots, d_n \rangle$, the abstract state can be written as the Cartesian product $\alpha(s) = A_1 \times \cdots \times A_n$ where $A_i = \{d_i\}$ if $v_i \in P$ and $A_i = D_i$ otherwise. The heuristic h^P induced by the projection to pattern P is called a *pattern database* (PDB).

Domain abstractions α can be defined as a tuple $\langle \alpha_1, \ldots, \alpha_n \rangle$ of *component abstractions* α_i of individual variables, where $\alpha_i : D_i \to 2^{D_i}$ denotes a partitioning of D_i such that $\alpha_i(d) \subseteq D_i$ represents all values mapped to the same abstract value by α_i. Like projections, domain abstractions are a special case of Cartesian abstractions: for state $s = \langle d_1, \ldots, d_n \rangle$, the abstract state can be written as the Cartesian product $\alpha(s) = \alpha_1(d_1) \times \cdots \times \alpha_n(d_n)$. Domain abstractions generalize projections: defining $\alpha_i(d_i) = \{d_i\}$ if $v_i \in P$ and $\alpha_i(d_i) = D_i$ otherwise yields a projection.

CEGAR for Domain Abstractions

In this section, we describe the CEGAR algorithm for computing a single domain abstraction. Recall that we need to be able to efficiently retrieve the pre-image of an abstraction $\alpha = \langle \alpha_1, \ldots, \alpha_n \rangle$ for the refinement step. This is possible as $\alpha^{-1} = \langle \alpha_1^{-1}, \ldots, \alpha_n^{-1} \rangle$. Furthermore, to obtain a compact representation of $\mathcal{T}^\alpha$, we use a perfect hash function to map (domain-abstracted) states to integers ranging from 0 to $|S^\alpha|$. This function can also be inverted efficiently. Combining these observations, we build our implementation of domain abstractions on top of the PDB implementation of Sievers, Ortlieb, and Helmert (2012), using their efficient implementations of $\mathcal{T}^\alpha$ and hashing.

[1] We treat abstract states as the set of states from the original transition system they represent, i.e., their pre-image under α.

Algorithm 1: CEGAR for domain abstraction generation.

Input: Planning task Π; subset of state variables *Blacklist*, variable *InitVar*
Output: Domain abstraction α

```
1:  function CEGAR(Π, Blacklist, InitVar)
2:      α ← INITIALIZE(Π, InitVar)
3:      while TIME() < MAXTIME do
4:          π ← COMPUTEPLAN(Π, α)
5:          // COMPUTEPLAN exits early if no abstract
               plan exists
6:          Flaws ← FINDFLAWS(Π, π, Blacklist)
7:          // FINDFLAWS exits early if concrete plan found
8:          if Flaws = ∅ then
9:              break
10:         ⟨α, Blacklist⟩ ← REFINE(α, Flaws, Blacklist)
11:     return α
```

Algorithm 2: Refinement of domain abstractions.

```
1: function REFINE(α, Flaws, Blacklist)
2:     (v ↦ d) ← SELECTFLAW(Flaws)
3:     α' ← α
4:     α'_v(d) = {d} // α'_v component abstraction for v
5:     if RESPECTSSIZELIMITS(α') then
6:         return ⟨α', Blacklist⟩
7:     else
8:         Blacklist = Blacklist ∪ {v}
9:         return ⟨α, Blacklist⟩
```

Algorithm 1 shows how to compute a single domain abstraction using CEGAR. It takes three arguments: a planning task Π and optionally a set of "blacklisted" variables *Blacklist* and a variable for initialization *InitVar*. For now, we assume *Blacklist* to be empty and *InitVar* to be unspecified.

At the start (line 2), the algorithm initializes the domain abstraction α via function INITIALIZE. Without a given *InitVar*, it computes the coarsest possible abstraction that maps all values of a variable to the same abstract value, i.e., $\alpha_i(d) = D_i$ for all $d \in D_i$ and all $1 \leq i \leq n$.

The main loop repeats until a user-specified time limit of MAXTIME is reached. In each iteration, the algorithm first computes an abstract plan π for the current abstraction using COMPUTEPLAN (line 4). We use the same algorithm as Rovner, Sievers, and Helmert (2019a) (Algorithm 1 in their technical report; Rovner, Sievers, and Helmert 2019b), using domain abstractions instead of PDBs, to compute *wildcard*-plans. These are defined as a sequence of steps where each

step corresponds to one transition from an abstract state to another, associated with all minimum-cost operators that induce this transition. If no abstract plan exists, the concrete task Π must be unsolvable.

Next, FINDFLAWS (line 6) attempts to execute π on Π. The algorithm is similar to Algorithm 2 by Rovner, Sievers, and Helmert (2019b) for finding flaws, however, we need to adapt their algorithm to compute sets of atoms rather than sets of variables. Our algorithm successfully executes a step of π on Π if any (random) action of the step can be applied, ignoring variables in *Blacklist*. If no action of a step can be applied, FINDFLAWS builds the union over the *precondition flaw(s)* of all actions a_i of the step. A precondition flaw of action a_i is the set $pre(a_i) \setminus s$ which contains the atoms of the precondition that are inconsistent with state s. Otherwise, i.e., if applicable actions exist for all steps of π, we distinguish two cases. If π did not lead to a state satisfying G of Π, FINDFLAWS returns the *goal flaw(s)* $G \setminus s$. Otherwise, if the blacklist is empty, π is a valid plan for Π and the planner exits. If the blacklist is not empty, FINDFLAWS returns an empty set of flaws which causes CEGAR to stop (line 9) because α cannot be refined further.

The final step of each iteration is to refine α (line 10) using REFINE shown in Algorithm 2. It selects one $v \mapsto d$ from *Flaws* using SELECTFLAW (line 2) and repairs it by updating the component α_v of α to map d to the set consisting of only itself (line 4). This ensures that the failed precondition or goal check cannot fail in α' anymore. However, if the updated α' violates a user-specified maximal number of abstract states, the flaw is addressed instead by adding v to *Blacklist*.[2] As FINDFLAWS ignores blacklisted variables, v cannot cause flaws in future iterations anymore.

We remark that in principle, we could also select and repair multiple flaws at once. However, initial experiments showed that this is not beneficial, mostly because abstractions grow too fast. Instead, we decided to stick with selecting a single flaw, which is also the approach used by CEGAR for computing Cartesian abstractions (Seipp and Helmert 2018). To select a single flaw, we consider two strategies: RAND chooses a flaw from *Flaws* uniformly at random, and MINGROWTH corresponds to the strategy "max-refined" by Seipp and Helmert (2018) which chooses a flaw such that its corresponding variable is most refined among all candidates. The potential benefit of MINGROWTH is that it causes the smallest increase in abstraction size and thus allows more refinements overall. In other words, MINGROWTH chooses v_i such that α_i has the largest domain among all i, breaking ties by choosing one of the candidates uniformly at random.

Algorithm 1 always terminates: in each refinement, it either separates a value of a variable or adds the variable to *Blacklist*. Either way, this ensures that the same flaw will not be found in any future iteration. Since the number of flaws is bound by the number of atoms, CEGAR terminates in at most $\prod_{i=1}^{n} |D_i|$ iterations assuming Π has n variables. Further, each operation in the CEGAR loop can be implemented to run in polynomial time in the representation size of Π and the size bound on abstractions.

[2]Note that blacklisting *atoms* instead of variables does not make sense because repairing *any* flaw (by splitting the atom) on the variable would violate the abstraction size limit.

Collections of Domain Abstractions

While domain abstractions do not suffer as badly from exponential growth as PDBs, we still cannot expect that single domain abstractions are informed enough to yield strong heuristics. Instead, we want to compute a diverse set of domain abstractions to combine them using state-of-the-art *saturated cost partitioning* (SCP) (Seipp, Keller, and Helmert 2020). We remark that domain abstractions, like PDBs and Cartesian abstractions, are well-suited to be used in SCPs: we can compute the minimum saturated cost function efficiently by iterating over all abstract transitions (Seipp, Keller, and Helmert 2020).

We follow Rovner, Sievers, and Helmert (2019a) to compute a collection of abstractions by repeatedly calling CEGAR (Algorithm 1). Their Algorithm 3 (Rovner, Sievers, and Helmert 2019b) repeatedly iterates over all goal variables of the task and restricts each call of CEGAR to compute a pattern starting from and containing only that goal variable. It further performs duplicate detection on generated patterns. When no progress is made (stagnation) for some time, it additionally forbids random subsets of non-goal variables to be included in the computed pattern using the *Blacklist* parameter. The algorithm has a time limit and a size limit for the computed collection.

As there are many more different domain abstractions than patterns, we expect to only rarely encounter duplicate domain abstractions (and also confirmed this experimentally). We therefore adapt the algorithm to drop the notion of stagnation and instead start blacklisting random subsets of variables[3] after a specified percentage of the total time limit has passed.

Furthermore, to better make use of the more fine-grained domain abstractions, we do not simply pass a different goal variable in each iteration, but consider different strategies for initializing the domain abstraction in each CEGAR run. Recall that INITIALIZE of Algorithm 1, when not given a variable via *InitVar*, initializes the domain abstraction α as the trivial abstraction. We denote this variant N for *no initialization*. Our collection algorithm additionally considers passing a *random goal variable* (G) or a *random arbitrary variable* (A). When given a variable *InitVar* $= v$, INITIALIZE considers two strategies: the *identity strategy* (I) means to perfectly represent the variable as in a projection to $\{v\}$, i.e., $\alpha_v(d) = \{d\}$ for all $d \in D_v$, and the *value strategy* (V) only splits off a single value d from the domain of v, i.e., $\alpha_v(d) = \{d\}$ and $\alpha_v(d') = D_v \setminus \{d\}$ for all $d' \neq d$, where d is the single goal value of v if $v \in Vars(G)$ and a random value in D_v otherwise.

Experiments

We implemented domain abstractions in Fast Downward version 22.06 (Helmert 2006). Experiments were conducted

[3]*Not* restricted to *non-goal* variables because we do not restrict CEGAR to a single goal variable as explained below.

0s	25s	50s	75s	100s
1058.8±7.2	1051.5±4.0	1049.0±3.6	1047.3±5.3	1045.1±5.1

Table 1: Coverage of different starting times for blacklisting. 0s denotes that blacklisting is enabled from the beginning and 100s denotes that no blacklisting is used.

on Intel Xeon Silver 4114 2.2 GHz processors using Downward Lab (Seipp et al. 2017) with a time limit of 30 minutes and a memory limit of 3.5 GiB for each run. Our benchmark set consists of 1827 problems from the optimal tracks of international planning competitions 1998–2018. All code, benchmarks, and experiment data are published online (Kreft et al. 2023).

We run our collection algorithm for 100s and combine the obtained abstractions using offline SCP heuristics with hybrid-optimized greedy orders diversified for 200s (Seipp 2017). We compare against the best pattern collection generation method by Rovner, Sievers, and Helmert (2019a) (limit of 100s, 1M states for individual PDBs, 10M states for the collection) and against Cartesian abstractions (limit of 1M transitions). Since these algorithms depend on randomness, we run our experiments 10 times with different random seeds to cancel out noise and report average values with standard deviations.

Before looking into different settings for refinement strategies, we evaluate the influence of blacklisting and abstraction sizes. To do so, we imitate PDBs in terms of abstraction initialization by using strategy GI, which means to initialize each CEGAR run with a single random goal variable (G) and abstracting it with full precision (I). At this stage, we fix flaws using the RAND strategy.

In contrast to PDBs that activate blacklisting after 20s of stagnation or after 75s of total construction time (Rovner, Sievers, and Helmert 2019a), we drop the stagnation strategy and test to enable blacklisting earlier or later. Table 1 shows that blacklisting from the beginning yields the best results, most likely because it increases the diversity of the set of abstractions significantly, which benefits SCP. We choose this setting for the remaining experiments.

Table 2 reports the number of solved tasks for different choices for the size limits. Rovner, Sievers, and Helmert recommend 1M states per individual abstraction and 10M states in the abstraction collection. We observe that coverage increases when reducing the single abstraction size limit to 100k. However, also reducing the collection size limit to maintain the ratio of 10 between abstraction and collection size is not beneficial. Instead, the ratio 100 proves even more effective when reducing the size limits further to 10k per abstraction and 1M for the collection. This makes sense because domain abstractions are more fine-grained than PDBs and therefore smaller abstractions can suffice to yield similarly good heuristics. We choose this setting for the remaining experiments.

Next, we turn our attention to different strategies for initializing and refining domain abstractions. We evaluate all possible parameter combinations. Table 3 shows a pair-wise

1M		10M		
10k	100k	10k	100k	1M
1142.4±3.8	1041.8±6.3	1122.1±4.6	1137.9±5.6	1058.8±7.2

Table 2: Coverage of different limits on states per abstraction (bottom row in the header) and states in the abstraction collection (top row in the header).

		RAND					MINGROWTH					PDB	Cart.	coverage
		N	GI	GV	AI	AV	N	GI	GV	AI	AV			
RAND	N	–	12	14	18	**21**	8	12	12	11	14	**24**	**37**	1132.0±5.3
	GI	**25**	–	**18**	**17**	**28**	**20**	**14**	**19**	**18**	**22**	**28**	**38**	**1142.4**±3.8
	GV	**19**	10	–	17	**20**	14	12	12	13	17	**26**	**36**	1135.7±4.9
	AI	**23**	14	**22**	–	**19**	16	15	18	17	15	**30**	**40**	1139.8±5.0
	AV	17	13	14	14	–	13	13	14	16	14	**29**	**41**	1131.1±3.9
MINGROWTH	N	**30**	18	**24**	**24**	**29**	–	15	14	**19**	**19**	**30**	**42**	1134.1±4.5
	GI	**31**	**14**	**27**	**20**	**30**	**19**	–	**13**	**19**	**23**	**32**	**41**	1139.6±3.9
	GV	**29**	15	**25**	**20**	**28**	**16**	11	–	**21**	**20**	**30**	**41**	1137.2±2.6
	AI	**29**	17	**27**	**20**	**27**	13	11	13	–	**17**	**33**	**42**	1132.9±4.0
	AV	**22**	13	**20**	**21**	**23**	11	11	8	12	–	**30**	**39**	1129.5±5.3
PDB		22	13	19	14	19	15	9	13	12	16	–	**33**	1091.5±3.1
Cart.		11	7	10	10	10	8	8	7	7	8	13	–	1070.4±0.7

Table 3: Domain-wise comparison in terms of coverage of our collection algorithm (10 variants), PDBs and Cartesian abstractions. An entry in row x and column y denotes the number of domains in which x solves more tasks than y. It is bold if $(x, y) \geq (y, x)$. The rightmost column shows the total coverage of the configuration in the row.

domain comparison of coverage of all configurations, including PDBs and Cartesian abstractions (Cart.). An entry in row x and column y denotes the number of domains in which configuration x solves more tasks than configuration y. The rightmost column shows total coverage.

We first focus on the different techniques for domain abstractions. While total coverage is very close generally, the data shows that the configurations using MINGROWTH generally yield better results than RAND because they solve more tasks in more domains in most pair-wise comparisons. As explained before, MINGROWTH allows to perform more refinements before reaching abstraction size limits, which apparently pays off in practice. The only RAND strategy that beats the MINGROWTH ones is the one with GI initialization (in terms of total coverage only; they are identical in terms of the pair-wise domain comparison). In fact, GI performs best independent of the refinement strategy and solves more tasks in more domains than all other configurations. It corresponds exactly to the initialization of a PDB with a single random goal variable. While it makes sense to initialize with variables that are relevant to reach the goal, we are surprised that only splitting the goal value is inferior. This would keep the initial abstraction small and allows to only split more values of this goal variable if they occur in flaws.

Finally, we observe that all domain abstraction configurations solve more tasks than PDBs and Cartesian abstractions.

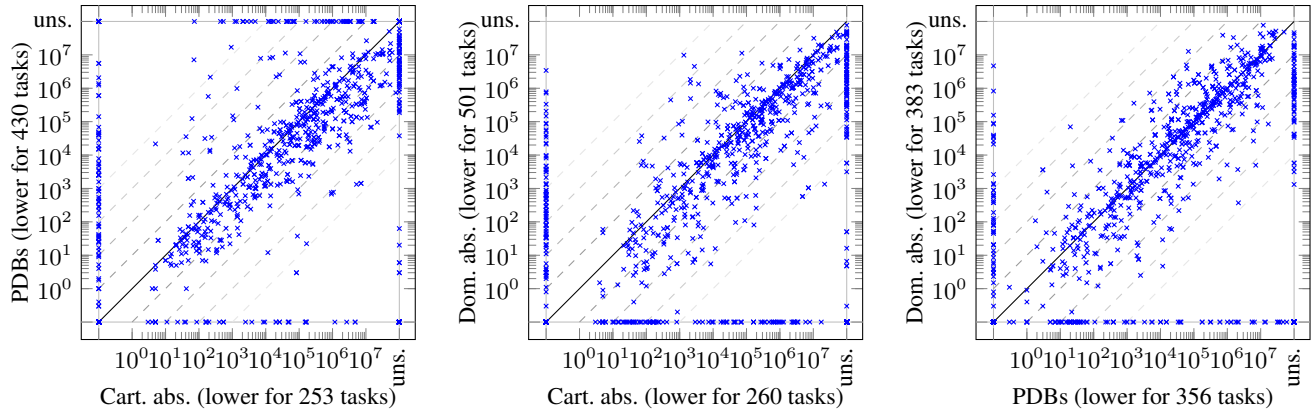

Figure 1: Number of expanded states before the last f-layer.

		original	+PDBs	+dom	+both
offl.	RAND	1141.8±1.3	**1167.8**±1.9	1158.4±4.0	1155.5±3.9
	MINGR			1167.4±4.1	1166.8±5.3
onl.	RAND	**1192.1**±1.3	**1192.1**±2.4	1172.5±3.2	1160.0±2.9
	MINGR			1181.2±5.3	1169.5±3.5

Table 4: Coverage of different configurations of the Scorpion planner: the IPC 2018 version using offline SCP (offl.) and the most recent version using online SCP (onl.), both computed using the abstractions originally used (original), original extended with CEGAR PDBs (+PDBs), original extended with our CEGAR domain abstractions (+dom), and original extended with both (+both).

In particular, our best configuration GI with RAND solves 50 tasks more than PDBs on average, which in turn solves 20 tasks more than Cartesian abstractions. Domain abstractions seem to find a good trade-off between abstraction granularity and efficiency in their computation.

To confirm that this is indeed the case, we also compare heuristic quality of the three classes of abstractions. Figure 1 shows the number of expanded states before the last f-layer for each pair, using RAND with GI to represent domain abstractions. Data points below the diagonal denote an advantage for the abstraction class annotated to the y-axis. The data points scatter in both directions from the diagonal by significant margins. We suspect these numbers are highly influenced by the success of SCP. Nevertheless, both PDBs and domain abstractions seem to outperform Cartesian abstractions in this comparison. Between PDBs and domain abstractions, the difference is less pronounced, but domain abstractions solve slightly more tasks with fewer expansions.

Finally, we also compare against Scorpion (Seipp 2018), a state-of-the-art planner based on abstraction heuristics. It also uses SCP but combines abstractions constructed with different techniques to get the most out of their sometimes complementary strengths. Its IPC 2018 version uses projections computed with hill climbing (Haslum et al. 2007), all interesting patterns of size ≤ 2 (Pommerening, Röger, and Helmert 2013), and Cartesian abstractions as we reported them above, combined in SCP heuristics computed offline before search for up to 200s. The most recent version of Scorpion computes subset-saturated SCPs (Seipp and Helmert 2019) online during search (Seipp 2021) and drops the hill climbing projections from the considered abstractions. Table 4 shows coverage for both versions of Scorpion, using offline SCPs (offl.) and online SCPs (onl.). Similar to Rovner, Sievers, and Helmert (2019a), we consider Scorpion using its original abstractions as explained above (original) and add CEGAR PDBs (+PDBs), our CEGAR domain abstractions (+dom), or both (+both) to the abstractions used by Scorpion. Since both RAND and MINGROWTH showed similar performance when using the best initialization strategy GI, we consider both of them for our domain abstractions.

The results for both offline and online SCPs make clear that there is an advantage to refine using the MINGROWTH strategy rather than RAND. Moreover, we can see that adding domain abstractions to the mix performs equally well as adding PDBs with offline SCPs, but less so with online SCPs. Adding both, however, does not improve the results. One possible explanation is that domain abstractions, being between PDBs and Cartesian abstractions, cannot add many better-informed abstractions to the mix.

Conclusions

We presented an algorithm based on the CEGAR principle for computing domain abstractions. We showed how to compute diverse collections using this algorithm as subroutine. Our experiments show that domain abstractions can perform better than collections of PDBs or Cartesian abstractions. We conclude that they offer a good trade-off between the coarser but faster PDBs and more fine-grained but more expensive Cartesian abstractions. Promising directions for future work include investigating further strategies for diversification and finding more "duplicates" by introducing a similarity measure for domain abstractions.

Acknowledgments

We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639). Moreover, this research was partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under grant agreement no. 952215.

References

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11(4): 625–655.

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2000. Counterexample-Guided Abstraction Refinement. In Emerson, E. A.; and Sistla, A. P., eds., *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, 154–169.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Edelkamp, S. 2001. Planning with Pattern Databases. In Cesta, A.; and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

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

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; Haslum, P.; Hoffmann, J.; and 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.

Hernádvölgyi, I. T.; and Holte, R. C. 2000. Experiments with Automatically Created Memory-Based Heuristics. In Choueiry, B. Y.; and Walsh, T., eds., *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA 2000)*, volume 1864 of *Lecture Notes in Artificial Intelligence*, 281–290. Springer-Verlag.

Kreft, R.; Büchner, C.; Sievers, S.; and Helmert, M. 2023. Code, Benchmarks and Experiment Data for the ICAPS 2023 Paper "Computing Domain Abstractions for Optimal Classical Planning with Counterexample-Guided Abstraction Refinement". https://doi.org/10.5281/zenodo.7733329.

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

Pommerening, F.; Röger, G.; and 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)*, 2357–2364. AAAI Press.

Rovner, A.; Sievers, S.; and Helmert, M. 2019a. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 362–367. AAAI Press.

Rovner, A.; Sievers, S.; and Helmert, M. 2019b. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning: Additional Material. Technical Report CS-2019-002, University of Basel, Department of Mathematics and Computer Science.

Seipp, J. 2017. Better Orders for Saturated Cost Partitioning in Optimal Classical Planning. In Fukunaga, A.; and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 149–153. AAAI Press.

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

Seipp, J. 2021. Online Saturated Cost Partitioning for Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 317–321. AAAI Press.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Seipp, J.; and Helmert, M. 2019. Subset-Saturated Cost Partitioning for Optimal Classical Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 391–400. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research*, 67: 129–167.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Sievers, S.; and Helmert, M. 2021. Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems. *Journal of Artificial Intelligence Research*, 71: 781–883.

Sievers, S.; Ortlieb, M.; and 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.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 105–111. AAAI Press.