Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning

Alexander Rovner, Silvan Sievers, Malte Helmert
University of Basel
Basel, Switzerland
alex.rovner@stud.unibas.ch, {silvan.sievers,malte.helmert}@unibas.ch

Abstract

We describe a new algorithm for generating pattern collections for pattern database heuristics in optimal classical planning. The algorithm uses the counterexample-guided abstraction refinement (CEGAR) principle to guide the pattern selection process. Our experimental evaluation shows that a single run of the CEGAR algorithm can compute informative pattern collections in a fairly short time. Using multiple CEGAR algorithm runs, we can compute much larger pattern collections, still in shorter time than existing approaches, which leads to a planner that outperforms the state-of-the-art pattern selection methods by a significant margin.

Introduction

Heuristics based on *pattern databases* (PDBs) have originally been introduced for solving the 15-puzzle (Culberson and Schaeffer 1998). Since then, they were successfully used for various other combinatorial problems and also adapted for domain-independent planning (e.g., Edelkamp 2001; 2002), where they are primarily used in an A* search to optimally solve classical planning tasks.

One inherent problem with PDBs is that they grow exponentially in the number of variables included in their patterns. Consequently, single PDB heuristics are usually not enough to produce strong heuristics. Instead, existing PDB-based techniques make use of *pattern collections*, possibly exploiting independence between patterns. One such example is the *canonical PDB* (CPDB) heuristic (Haslum et al. 2007) that sums PDB heuristic values whenever this is known to be admissible and computes their maximum otherwise. More recently, *cost-partitioning* (CP) heuristics further pushed the performance of classical planners by allowing to admissibly combine arbitrary heuristics, and in particular abstraction heuristics such as PDBs, structural patterns or Cartesian abstractions (e.g., Katz and Domshlak 2010; Pommerening, Röger, and Helmert 2013; Seipp, Keller, and Helmert 2017a; Seipp and Helmert 2018).

Besides the question of how to combine PDB heuristics, the most important question is how to select "good" pattern collections. The first work in planning that considered this problem cast it as an optimization problem that aimed at maximizing the mean heuristic value of a greedy zero-one cost partitioning heuristic over the PDBs induced by the collection (Edelkamp 2006). Greedy zero-one cost partitioning considers the PDBs in some order and attributes the full cost of each operator to the first PDB affected by that operator, treating it as zero-cost for all other PDBs. To solve the optimization problem, Edelkamp used a genetic algorithm.

Haslum et al. (2007) used hill-climbing (HC) in the space of pattern collections to minimize the number of expansions of a heuristic search using the CPDB heuristic for the given pattern collections, based on a model of IDA* runtime by Korf et al. (2001). Franco et al. (2017) recently revisited the problem in an approach called complementary PDB creation (CPC) that modifies the genetic algorithm of Edelkamp to minimize a sampling-based estimate of search tree size (Chen 1992; Lelis, Zilles, and Holte 2013).

In this work we suggest another approach for pattern selection. Our main motivation is the observation that both HC for optimizing CPDB heuristics and CPC are rather slow approaches that require a time limit to stop optimization and switch to search. In contrast, we want to devise a fast method for generating pattern collections.

To this end, we use the *counterexample-guided abstraction refinement* (CEGAR) principle (Clarke et al. 2000), which has successfully been used in the context of abstraction heuristics for classical planning for the case of Cartesian abstractions (Seipp and Helmert 2018). CEGAR for generating PDBs iteratively refines a pattern collection by only introducing variables relevant to the collection. Since patterns cannot grow indefinitely, the algorithm terminates quickly while being able to generate reasonably good collections.

In an experimental study we first evaluate the CEGAR algorithm to generate a single disjoint pattern collection, with the rationale that disjoint pattern collections are "close to additive", which can be leveraged by CP techniques like *saturated cost partitioning* (SCP) (Seipp and Helmert 2014; Seipp, Keller, and Helmert 2017b). In a second step, we further restrict the CEGAR algorithm to generate single patterns and run it multiple times to form a larger pattern collection. We discuss how to ensure that CEGAR runs produce diverse patterns, which is crucial to the success of cost partitioning. The resulting approach significantly outperforms the state of the art in pattern selection.

Background

Classical planning. We consider planning tasks with SAS$^+$-style representations (Bäckström and Nebel 1995; Helmert 2009). A task is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$. $\mathcal{V}$ is a finite set of *variables* v, each with a finite *domain* $\mathit{dom}(v)$. *Partial states* s are defined on a subset $\mathit{vars}(s) \subseteq \mathcal{V}$ of variables and map each $v \in \mathit{vars}(s)$ to a value in its domain, written $s[v]$. If $\mathit{vars}(s) = \mathcal{V}$, s is called a *state*. Partial states s and s' are *consistent* if $s[v] = s'[v]$ for all $v \in \mathit{vars}(s) \cap \mathit{vars}(s')$. $\mathcal{O}$ is a finite set of *operators* $o = \langle \mathit{pre}(o), \mathit{eff}(o), c(o) \rangle$, where $\mathit{pre}(o)$ and $\mathit{eff}(o)$ are partial states called *precondition* and *effect*, and $c(o) \in \mathbb{R}_0^+$ is called the *cost* of o. s_0 is the *initial state* and $s_\star$ is a partial state called the *goal*.

A planning task induces a transition system which has a transition from state s to state t labeled with o if o is *applicable* in s, i.e., $\mathit{pre}(o)$ and s are consistent, and t is the successor state of s and o, i.e., $t[v] = \mathit{eff}(o)$ for all $v \in \mathit{vars}(\mathit{eff}(o))$ and $t[v] = s[v]$ for all $v \notin \mathit{vars}(\mathit{eff}(o))$. The objective of classical planning is to find a *plan*, i.e., a sequence of operators $\pi = \langle o_1, \dots, o_n \rangle$, where each o_i is successively applicable starting from s_0 and which leads to a state consistent with $s_\star$. The cost of such a plan is $\sum_{i=1}^{n} c(o_i)$. We deal with *optimal* planning, where the objective is to find plans of minimal cost or to prove that no plan exists.

PDBs and Combination Techniques. To solve planning tasks optimally, we use the A* algorithm (Hart, Nilsson, and Raphael 1968) with an *admissible heuristic*. A heuristic h maps states s to numerical estimates of $h^*(s)$, the optimal solution cost from s. A heuristic is admissible if $h(s) \leq h^*(s)$ and *perfect* if $h(s) = h^*(s)$ for all states. We consider *abstraction heuristics* and in particular *pattern database (PDB)* heuristics. A PDB heuristic h^P is induced by a pattern $P \subseteq \mathcal{V}$ of the variables $\mathcal{V}$ of a planning task Π. It is defined as the perfect heuristic for $\Pi|_P$, which is the task Π projected onto P. $\Pi|_P$ can be computed by removing all occurrences of variables $v \notin P$ from Π.

To combine multiple patterns, we use two techniques. The first is the *canonical PDB (CPDB)* heuristic (Haslum et al. 2007), which is based on a simple sufficient, but not necessary *additivity* criterion for PDBs. Two PDBs P, Q are additive if there exist no variables $v \in P, w \in Q$ such that $v, w \in \mathit{vars}(\mathit{eff}(o))$ for any operator o. The CPDB heuristic for a collection of patterns C sums the heuristic values of PDBs induced by patterns of maximal additive subsets of C and takes the maximum over the resulting values.

The second technique is *saturated cost partitioning* (SCP) (Seipp and Helmert 2013), which has been successfully used for Cartesian abstractions and PDB heuristics (Seipp, Keller, and Helmert 2017a). Like greedy zero-one cost partitioning, SCP greedily distributes the costs among the heuristics in a given order, but in a more intelligent way that avoids using up costs that do not contribute to the heuristic value.

Disjoint Pattern Collections with CEGAR

The first pattern generation algorithm we propose computes a disjoint pattern collection, i.e., a set of patterns where no two patterns in the set share a state variable. Disjoint collections are a compromise between single patterns (which often do not lead to good heuristics) and arbitrary collections (which form a much more complex space to search in).

Algorithm 1 CEGAR for pattern collection generation.

```
Input: Planning task $\Pi$; subset of state variables Blacklist
Output: Pattern collection $C$
 1: function CEGAR($\Pi$, Blacklist)
 2:     $C \leftarrow \{\{v\} \mid v \in \mathit{vars}(s_\star)\}$ // with $s_\star$ the goal of $\Pi$
 3:     Plans $\leftarrow \{\}$ // maps patterns to abstract plans
 4:     while TIME() $<$ MAXTIME do
 5:         for $P \in C$ s.t. Plans[$P$] is undefined do
 6:             Plans[$P$] $\leftarrow$ COMPUTEPLAN($\Pi, P$)
 7:             // COMPUTEPLAN exits early if no abstract
                  plan exists
 8:         Flaws $\leftarrow \{\langle P, v\rangle \mid P \in C,$
                  $v \in$ FINDFLAWS($\Pi$, Plans[$P$], Blacklist)$\}$
 9:         // FINDFLAWS exits early if concrete plan found
10:         if Flaws $= \emptyset$ then
11:             break
12:         $\langle C$, Blacklist$\rangle \leftarrow$ REFINE($C$, Flaws, Blacklist)
13:     return $C$
```

Algorithm 2 REFINE of CEGAR.

```
 1: function REFINE($C$, Flaws, Blacklist)
 2:     $\langle P, v\rangle \leftarrow$ SELECTFLAWUNIFORMLY(Flaws)
 3:     if $v \in P'$ for some $P' \in C$ with $P' \neq P$ then
 4:         $C' \leftarrow (C \setminus \{P, P'\}) \cup \{P \cup P'\}$
 5:     else
 6:         $C' \leftarrow (C \setminus \{P\}) \cup \{P \cup \{v\}\}$
 7:     if RESPECTSSIZELIMITS($C'$) then
 8:         return $\langle C'$, Blacklist$\rangle$
 9:     else
10:         return $\langle C$, Blacklist $\cup \{v\}\rangle$
```

We follow the CEGAR principle to guide pattern selection. Algorithm 1 shows pseudo-code for the main procedure, which receives two arguments: the planning task Π and a set of "blacklisted" state variables *Blacklist*. For now, assume that the empty set is passed into *Blacklist*. We will consider variations in the following section.

The algorithm keeps track of the current (disjoint) pattern collection C, which is initialized to a set of singleton patterns, one for each goal variable of Π (line 2), as in the HC algorithm by Haslum et al. (2007). Variable *Plans* keeps track of an *optimal abstract plan* for each pattern P in the collection. Note that we only compute *one* abstract plan for each pattern throughout the algorithm (lines 5–7).

In each iteration of the main loop, we look for *flaws* of the abstract plans, i.e., reasons why the abstract plans for the patterns in C do not solve Π. Flaws are computed separately for each pattern in C (line 8). Each flaw is represented as a pair $\langle P, v\rangle$ where $P \in C$ is a pattern and $v \notin P$ is a variable such that the absence of v from P causes a violation of a precondition or goal condition in the abstract plan for P. We then use these flaws to refine the pattern collection (line 12). The CEGAR loop continues until a time limit is reached (using parameter MAXTIME; line 4) or no more flaws exist.

Computing Abstract Plans. COMPUTEPLAN(Π, P) operates the abstract planning task $\Pi|_P$. We refer to Rovner, Sievers, and Helmert (2019) for pseudo-code. The function first checks if $\Pi|_P$ is solvable. If not, CEGAR terminates immediately because this proves that Π is also unsolvable. Otherwise, we compute an optimal abstract plan. We consider two kinds of abstract plans, with the choice between the two kinds controlled by an algorithm parameter.

The first kind is a *regular plan*, i.e., the usual form of plan for a planning task, represented as a sequence of operator labels. This is a highly committing form of abstract plan. Projections of planning tasks tend to have many parallel state transitions (multiple transitions between the same pair of states labeled by different operators), and at the abstract level there is no way to decide which of these is more likely to lead to a successful plan in the concrete state space.

Therefore, as an alternative kind of abstract plan we consider *wildcard plans*, where we commit to a single sequence of abstract states, but leave the choice of which exact operators to apply in each step open. Mathematically, we represent wildcard plans as a sequence of steps, where each step corresponds to one abstract state transition and is associated with all minimum-cost operators that induce this state transition. The operators within each step are represented as a sequence with uniformly random order. (The order matters for function FINDFLAWS, described below.)

Besides the form of abstract plan, an important decision is *which* abstract plan to compute because there are often many different optimal abstract plans to choose from, even with the less committing wildcard plans. To compute an optimal abstract plan, we first compute the perfect heuristic for the projection (i.e., the pattern database for P), which will be needed anyway. We then extract a plan by performing the equivalent of an A* search in the abstract space that considers the successors of every state in uniformly random order and breaks ties when expanding nodes in favor of low h^*-values primarily and using FIFO secondarily.

With a perfect heuristic, this can be implemented efficiently as an enforced hill-climbing search (Hoffmann and Nebel 2001) using h^* as the heuristic and pruning all states whose f-value does not equal $h^*(s_0)$. To faithfully capture A* tie-breaking, we make one minor modification to the enforced hill-climbing algorithm: when expanding a state with an improving successor, we pick a *best* improving successor (uniformly randomly from all best improving successors) rather than the first improving successor. In planning tasks without zero-cost operators, this can be simplified further, as the breadth-first aspect of enforced hill-climbing is never needed: every non-goal state has an improving successor.

Finding Flaws. Function FINDFLAWS$(\Pi, \pi, \mathit{Blacklist})$ attempts to execute the abstract plan π (regular or wildcard) in the concrete planning task Π and returns the (possibly empty) set of flaws which prevented executing π. In this context, a flaw is a state variable v of Π. We only discuss wildcard plans: regular plans can be viewed as a special case where every plan step consists of one operator. We again refer to Rovner, Sievers, and Helmert (2019) for pseudo-code.

The third input besides the planning task and abstract plan is a set of "blacklisted" state variables. Flaws involving blacklisted state variables are ignored. The main idea behind blacklists is to allow the CEGAR process to continue when space limitations prevent addressing certain flaws, but they can also be used for diversification of the generated patterns. More on both uses of blacklists below.

We attempt to execute π using a simple greedy approach. A wildcard plan step (= sequence of operators) is applicable in state s of Π if any of its operators is applicable in s, where preconditions on blacklisted variables are ignored. Applying the plan step means applying the first such applicable operator in s, leading to the successor state s'. We execute the plan steps in π in sequence until we hit an inapplicable plan step or until all plan steps have been applied. In the former case, we collect all non-blacklisted variables in the failing plan step with violated preconditions and return these as the flaws responsible for the failure of π. In the latter case, the flaws are all non-blacklisted variables of Π with violated goal conditions after executing π. If all goal conditions are satisfied, π executed successfully. In this case we check if it would also execute successfully with no blacklisted variables. If yes, π represents an optimal solution to Π, and we exit without returning to the CEGAR loop. If no, we return the empty set of flaws but continue the CEGAR loop. (Other flaws may exist for other patterns in the collection.)

We remark that less greedy approaches are possible: we might backtrack over the choice points in executing a wildcard plan, or we might consider multiple abstract plans. However, this is computationally much more challenging. In particular, it is easy to see that it is NP-complete to test if a given wildcard plan has an executable instantiation and PSPACE-complete to test if there exists an optimal abstract plan that is also a concrete plan.

Repairing a Flaw. The final ingredient of the CEGAR algorithm is function REFINE$(C, \mathit{Flaws}, \mathit{Blacklist})$, shown in Algorithm 2. Its task it is to address one of the identified flaws, which is selected uniformly randomly (line 2).

Recall that a flaw in this context is a pair $\langle P, v\rangle$ such that the plan for P could not be executed because of a violation of a (pre- or goal) condition on variable $v \notin P$. Such a flaw can be addressed by adding v to P (line 6). However, if v is already present in some other pattern $P' \in C$, we cannot add it to P directly without violating the disjointness of C. Therefore, in this case the suggested repair is to merge P and P' into a single pattern (line 4).

In either case, the suggested repair might be impossible because the resulting pattern collection would exhaust memory limits. We enforce a maximal number of abstract states for each individual PDB and a maximal number of abstract states across all pattern databases. If the new collection would violate either limit, we abstain from updating C and instead eliminate the flaw by adding v to the blacklist.

Runtime Analysis. It is easy to see that, even without time and memory limits, the CEGAR loop would eventually terminate because each iteration replaces some pattern in C by a larger pattern or adds a variable to the blacklist.

Formally, define the *potential* of pattern collection C and blacklist B as $\mathit{pot}(C, B) = \sum_{P \in C} |P| - |C| + |B|$, i.e., the total number of variables in all patterns minus the number of patterns plus the number of blacklisted variables. At the beginning of the algorithm with all-singleton patterns, we have $\mathit{pot}(C, B) = |B| \geq 0$, and at all times we have $\mathit{pot}(C) \leq 2|\mathcal{V}| - 1$, with the maximum reached if all variables are present in the collection, there is only a single pattern, and all variables are blacklisted. Each iteration of the loop increases the potential by 1 (by adding a variable to one of the patterns, thus increasing $\sum_{P \in C} |P|$, by merging two patterns, thus decreasing $|C|$, or by adding a variable to the blacklist, thus increasing $|B|$). Therefore, the total number of loop iterations is bounded by $2|\mathcal{V}| - 1$.

We abstain from giving precise runtime bounds for each of the individual operations within the CEGAR loop because this would require more implementation details. Even so, it is clear that all operations can be implemented in time that is polynomial in the representation size of Π and the bound on the total number of abstract states.

Multiple CEGAR Runs

A single disjoint pattern collection often does not lead to a strong heuristic. We therefore also consider running the CEGAR algorithm multiple times, combining all generated patterns into an overall (usually non-disjoint) collection. Rovner, Sievers, and Helmert (2019) shows pseudo-code.

Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be the input task. With multiple CEGAR runs, there is no need for a single run to compute multiple patterns. Indeed, it may be desirable for each run to produce a single pattern, so that it can complete as quickly as possible and in particular avoid the costly merging of patterns. Therefore, we perform each CEGAR run on a modified task derived from Π by restricting the goal to a single variable. (This task behaves exactly like Π, but has a larger set of goal states.) The only modification of the CEGAR algorithm necessary is that we can no longer exit the planning algorithm completely when CEGAR finds a solution for its planning task (the "early exit" in FINDFLAWS) because that solution might not solve the original planning task.

The overall algorithm works as follows. We compute a random permutation of the goal variables, over which we iterate indefinitely in cyclic order until a time limit is reached. Each iteration applies CEGAR to Π restricted to the current goal and adds the resulting pattern to the overall collection of patterns. (Note that CEGAR always returns a single pattern on a single-goal task.) Despite the randomness in the CEGAR algorithm, with many CEGAR runs we may of course produce the same pattern multiple times. Therefore, we allow setting a *stagnation* time limit: if no new pattern is added to the collection for a certain time, the algorithm terminates.

For further diversification of patterns, we also consider algorithm variants where the calls to CEGAR pass in a random blacklist. Let $V' = \mathcal{V} \setminus \mathit{vars}(s_\star)$. If *random blacklisting* is enabled, in each CEGAR run we select a number $K \in \{0, \ldots, |\mathcal{V}| - 1\}$ uniformly randomly and then select a blacklist of K state variables, chosen uniformly randomly from V'. (If $K > |V'|$, we select all variables in V'.)

We consider two algorithm variants with blacklisting. In both variants, blacklisting is initially disabled. In the first variant, it is enabled after a given amount of time. In the second, it is enabled when the stagnation limit triggers, i.e., at the point when stagnation would normally terminate the algorithm. In this setting, we reset the stagnation timer when blacklisting is enabled and terminate the algorithm when stagnation triggers a second time.

Experiments

We implemented both CEGAR algorithms in Fast Downward (Helmert 2006). We limited pattern construction time to $100s$ and the number of states to 1M for individual PDBs and 10M for all PDBs.

We include a simple randomized baseline for the CEGAR algorithm, which we call (single) *randomized causal graph* (sRCG). It computes a single pattern by performing a random walk on the causal graph, starting from a random goal variable, and adding each variable visited by the walk. Since the CEGAR algorithm only ever adds variables to a pattern that are connected to the existing pattern in the causal graph, sRCG can be viewed as a "blind" variant of the CEGAR algorithm not guided by flaws. Analogously to how the multiple CEGAR algorithm works, *mRCG* repeatedly runs sRCG to compute a pattern and thus serves as a blind variant of the multiple CEGAR algorithm. As a state-of-the-art approach for computing a single pattern, we also implemented the Gamer (G) algorithm (Kissmann and Edelkamp 2011) to compute a PDB with at most 10M states (time limit $900s$).

As state-of-the-art pattern collection generation techniques, we consider the following three variants: the systematic generation of patterns (Pommerening, Röger, and Helmert 2013) up to size 2 (SYS), the HC algorithm as implemented in Fast Downward (Sievers, Ortlieb, and Helmert 2012), and the CPC algorithm. We run both HC and CPC with a $100s$ time limit like our method, and also with a $900s$ time limit which is commonly used in the literature (for CPC, the resulting configuration is identical to CPC-E in the work of Franco et al., 2017), indicated by appending 1 or 9 to the algorithm name.

We evaluate all techniques (except the single PDBs sRCG and G) using the CPDB heuristic and the SCP heuristic, the latter using general costs and greedy computation of hybrid-optimized orders for $200s$ as in the work by Seipp (2017). All planners are run on all benchmarks from the optimal tracks of all IPCs, a set consisting of 1827 tasks across 65 domains. Each run is limited to $1800s$ and 3.5 GiB. Experiments were run on Intel Xeon Silver 4114 CPUs, using Downward Lab (Seipp et al. 2017). All randomized algorithms were run 10 times, and the reported results are averages over these 10 runs. For all results reported below, the technical report (Rovner, Sievers, and Helmert 2019) contains additional per-domain results.[1]

Disjoint Pattern Collections

We first evaluate the basic CEGAR algorithm (Algorithm 1) to generate a disjoint pattern collection, comparing using a regular plan (regP) vs. computing wildcard plans (wcP).

[1]Implementation: https://doi.org/10.5281/zenodo.2628703, dataset: https://doi.org/10.5281/zenodo.2628693, benchmarks: http://doi.org/10.5281/zenodo.2616479.

		regP	wcP	SYS	HC1	CPC1	HC9	CPC9	sRCG	G
CPDB	Coverage	854.7	853.8	769	935.5	951.7	953.4	**975.7**	758.8	839
	C. time	0.35	0.27	**0.06**	1.77	103.56	2.38	910.65	0.07	5.49
SCP	Coverage	943.7	946.6	981	946.4	**1033.5**	965.4	1021.1		
	C. time	0.63	0.48	**0.05**	3.05	103.82	4.97	876.21		

Table 1: Coverage (sum) and construction time (geometric mean for tasks where all algorithms completed construction) of the single CEGAR algorithm (first two columns), SYS, HC, and CPC, with the CPDB (top) and SCP (bottom) heuristics, and sRCG and G (single PDB heuristic).

Table 1 shows aggregated coverage and runtime to construct pattern collections (excluding time to compute the SCP heuristic).

Considering coverage first, we observe that both of our methods are clearly better than the pattern computed by the simple randomized baseline (sRCG) and by Gamer. Using the more powerful wildcard plans is slightly better with the SCP heuristic but not with the CPDB heuristic. Comparing the heuristics more generally, it is clear that SCP is stronger than CPDBs in exploiting the same collection of PDBs, however the impact is different for different pattern selection methods. HC, which optimizes its collection for CPDBs, profits the least. With CPC, the improvement is large in particular when using a $100s$ limit, presumably because the collections generated by CPC grow too large when using a $900s$ limit. CEGAR profits even more from using the SCP heuristic, increasing coverage by 89 and 92.8. The best improvement from changing the heuristic is obtained with the SYS method, which is, however, mainly due to the fact that it performs very poorly (close to the randomized baseline) with the CPDB heuristic.

Compared to SYS, HC and CPC, the single CEGAR algorithm lags behind in coverage, but it constructs the pattern collection much faster than HC and CPC, though not faster than SYS. On average, CEGAR only reaches the $100s$ construction time limit on 7.25 tasks and solves 162.35 tasks during pattern construction.

Multiple CEGAR Runs

We now evaluate the multiple CEGAR algorithm. We again compare regular against wildcard plans (regP, wcP) and evaluate using the vanilla algorithm (v), terminating after stagnation for $20s$ (s), using blacklisting after $75s$ (b), and the combination of both, i.e., using blacklisting after the first stagnation (sb). We restrict the evaluation to the stronger SCP heuristic and use the best settings for HC ($900s$) and CPC ($100s$).

Table 2 shows a domain comparison in terms of coverage and includes aggregated coverage and construction time in the two rightmost columns. Comparing our different methods, we observe that stagnation (s) only slightly helps in terms of coverage, but greatly reduces construction time. Blacklisting (b), on the other hand, increases coverage, so the intended diversification works as expected. Combining both (sb) results in the best performance. Comparing regular vs. wildcard plans, there is clear preference for wildcard plans, which compare favorably on a per-domain basis against the regular plan variant.

		regP				wcP				mRCG	SYS	HC9	CPC1	total	c. t.
		v	s	b	sb	v	s	b	sb						
regP	v	–	**4**	6	6	11	11	12	11	**28**	**24**	**24**	24	1055.1	46.32
	s	**4**	–	9	7	10	9	12	11	**28**	**24**	**24**	24	1054.9	28.98
	b	**15**	**14**	–	2	**19**	**19**	11	10	**34**	**24**	**30**	**27**	1080.1	46.34
	sb	**16**	**15**	**6**	–	**20**	**19**	13	11	**34**	**24**	**30**	**27**	1081.1	37.43
wcP	v	**18**	**17**	18	16	–	**3**	7	7	**27**	**22**	**25**	**24**	1063.2	51.81
	s	**17**	**16**	18	17	2	–	7	8	**27**	**22**	**25**	**25**	1063.7	29.20
	b	**24**	**24**	**18**	**16**	**17**	**17**	–	3	**33**	**22**	**31**	**28**	1085.0	51.80
	sb	**25**	**25**	**18**	**17**	**18**	**17**	**6**	–	**33**	**22**	**31**	**28**	**1087.2**	39.65
mRCG		18	18	14	14	18	18	12	12	–	17	22	14	1018.7	10.07
SYS		20	20	19	19	20	20	18	18	**26**	–	15	20	981	**0.05**
HC9		21	20	15	15	19	19	13	13	**26**	**17**	–	17	965.4	5.00
CPC1		**26**	**26**	23	23	23	23	20	20	**32**	**23**	**28**	–	1033.5	103.85

Table 2: Domain comparison in terms of coverage of the multiple CEGAR algorithm (8 variants), mRCG, SYS, HC, and CPC with the SCP heuristic. 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 two rightmost columns show total coverage and pattern collection construction time (c. t.) of the algorithm in the row.

Compared against the previous results of using a single CEGAR algorithm run with the SCP heuristic, shown in Table 1, coverage is increased dramatically from the previous best value of 946.60 to 1087.2 (wcP+sb). This comes at the cost of an increased construction time, which is, however, still much lower than that of CPC. Furthermore, in terms of coverage, all of our methods outperforms the previous state-of-the-art pattern generation techniques, and our best configuration (wcP+sb) solves 53.7 tasks more than the previous best method CPC.

Comparison Against State of the Art

Finally, we also compare against two related state-of-the-art planners. The first is CPC with symbolic PDBs (CPC-S-P of Franco et al., 2017). The second is $h^{\text{SCP}}_{\text{hybrid-opt}}$ (Seipp, Keller, and Helmert 2017a), a planner using the same SCP heuristic as in this work, but computed over PDBs generated through HC and SYS and Cartesian abstractions generated with a CEGAR approach. As a modification of this planner, we include our best approach (multiple CEGAR, wcP+sb) to its set of sources for computing abstractions. We remark that CPC-S-P corresponds to Complementary2 (Franco, Lelis, and Barley 2018), the runner-up of IPC 2018, and $h^{\text{SCP}}_{\text{hybrid-opt}}$ corresponds to Scorpion (Seipp 2018), the planner from IPC 2018 performing best the largest number of domains, except that both IPC planners additionally use the h^2-based preprocessor by Alcázar and Torralba (2015).

Table 3 shows aggregated coverage. Our best planner outperforms the symbolic variant of CPC, which benefits from much larger PDBs due to their succinct symbolic representation. We thus believe that our techniques could be useful also in the context of symbolic PDBs. The planner $h^{\text{SCP}}_{\text{hybrid-opt}}$

	best	CPC-S-P	$h^{\text{SCP}}_{\text{hybrid-opt}}$	$h^{\text{SCP}}_{\text{hybrid-opt}}$+best
Coverage	1087.2	1073	1129.5	**1138.8**

Table 3: Coverage of our best method (best), CPC-S-P, $h^{\text{SCP}}_{\text{hybrid-opt}}$, and an extension of $h^{\text{SCP}}_{\text{hybrid-opt}}$ to also compute PDBs by using our best method.

is stronger than our best planner, but combining it with our best technique slightly improves its coverage even further.

Conclusions

We described a new algorithm for computing pattern collections using the CEGAR principle. We first evaluated using a single such CEGAR run to compute disjoint pattern collections, which already achieves promising results with very short pattern construction time. By embedding the CEGAR algorithm in an approach that repeatedly uses a CEGAR run to generate a single pattern and diversifying these patterns over different CEGAR runs, we obtained a fast method for generating large pattern collections. Computing the SCP heuristic over these PDBs resulted in state-of-the-art performance in pattern selection.

In future work, we would like to investigate further means of diversifying the resulting pattern collections. The approach could be extended to also include domain abstractions. Finally, instead of computing heuristics *after* the pattern selection process, the two could be combined, devising a heuristic construction method that interleaves pattern selection with cost partitioning.

Acknowledgments

This work was supported by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan).

References

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In *Proc. ICAPS 2015*, 2–6.

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

Chen, P. C. 1992. Heuristic sampling: A method for prediction the performance of tree searching programs. *SICOMP* 21(2):295–315.

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2000. Counterexample-guided abstraction refinement. In *Proc. 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 *Proc. ECP 2001*, 84–90.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proc. AIPS 2002*, 274–283.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *Proc. MoChArt 2006*, 35–50.

Franco, S.; Torralba, Á.; Lelis, L. H. S.; and Barley, M. 2017. On creating complementary pattern databases. In *Proc. IJCAI 2017*, 4302–4309.

Franco, S.; Lelis, L. H. S.; and Barley, M. 2018. The Complementary2 planner in the IPC 2018. In *IPC-9 planner abstracts*, 32–36.

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 *Proc. AAAI 2007*, 1007–1012.

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

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

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

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *Proc. AAAI 2011*, 992–997.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening A*. *AIJ* 129:199–218.

Lelis, L. H. S.; Zilles, S.; and Holte, R. C. 2013. Predicting the size of IDA*'s search tree. *AIJ* 196:53–76.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In *Proc. IJCAI 2013*, 2357–2364.

Rovner, A.; Sievers, S.; and Helmert, M. 2019. 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., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In *Proc. ICAPS 2013*, 347–351.

Seipp, J., and Helmert, M. 2014. Diverse and additive Cartesian abstraction heuristics. In *Proc. ICAPS 2014*, 289–297.

Seipp, J., and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *JAIR* 62:535–577.

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

Seipp, J.; Keller, T.; and Helmert, M. 2017a. A comparison of cost partitioning algorithms for optimal classical planning. In *Proc. ICAPS 2017*, 259–268.

Seipp, J.; Keller, T.; and Helmert, M. 2017b. Narrowing the gap between saturated and optimal cost partitioning for classical planning. In *Proc. AAAI 2017*, 3651–3657.

Seipp, J. 2017. Better orders for saturated cost partitioning in optimal classical planning. In *Proc. SoCS 2017*, 149–153.

Seipp, J. 2018. Fast Downward Scorpion. In *IPC-9 planner abstracts*, 77–79.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient implementation of pattern database heuristics for classical planning. In *Proc. SoCS 2012*, 105–111.