Pattern Selection for Optimal Classical Planning with Saturated Cost Partitioning

Abstract

Pattern databases are the foundation of some of the strongest admissible heuristics for optimal classical planning. Experiments showed that the most informative way of combining information from multiple pattern databases is to use saturated cost partitioning. Previous work selected patterns and computed saturated cost partitionings over the resulting pattern database heuristics in two separate steps. We introduce a new method that uses saturated cost partitioning to select patterns and show that it outperforms all existing pattern selection algorithms.

Introduction

A∗ search (Hart, Nilsson, and Raphael 1968) with an admissible heuristic (Pearl 1984) is one of the most successful methods for solving classical planning tasks optimally. An important building block of some of the strongest admissible heuristics are pattern database (PDB) heuristics. A PDB heuristic precomputes all goal distances in a simplified state space obtained by projecting the task to a subset of state variables, the pattern, and uses these distances as lower bounds on the true goal distances. PDB heuristics were originally introduced for solving the 15-puzzle (Culberson and Schaeffer 1996) and have later been generalized to many other combinatorial search tasks (e.g., Korf 1997; Felner, Korf, and Hanan 2004) and to the setting of domain-independent planning (Edelkamp 2001).

Using a single PDB heuristic of reasonable size is usually not enough to cover sufficiently many aspects of challenging planning tasks. It is therefore often beneficial to compute multiple PDB heuristics and to combine their estimates admissibly (Holte et al. 2006). The simplest approach for this is to choose the PDB with the highest estimate in each state. Instead of this maximization scheme, we would like to sum estimates, but this renders the resulting heuristic inadmissible in general. However, if two PDBs are affected by disjoint sets of operators, they are independent and we can admissibly add their estimates (Korf and Felner 2002; Felner, Korf, and Hanan 2004). Haslum et al. (2007) later generalized this idea by introducing the canonical heuristic for PDBs, which computes all maximal subsets of pairwise independent PDBs and then uses the maximum over the sums of independent PDBs as the heuristic value.

Cost partitioning (Katz and Domshlak 2008; Yang et al. 2008) is a generalization of the independence-based methods above. It makes the sum of heuristic estimates admissible by distributing the costs of each operator among the heuristics. The literature contains many different cost partitioning algorithms such as zero-one cost partitioning (Edelkamp 2002; Haslum et al. 2007), uniform cost partitioning (Katz and Domshlak 2008), optimal cost partitioning (Katz and Domshlak 2008; Karpas and Domshlak 2009; Katz and Domshlak 2010; Pommerening et al. 2015), post-hoc optimization (Pommerening, Röger, and Helmert 2013) and delta cost partitioning (Fan, Müller, and Holte 2017).

Seipp, Keller, and Helmert (2017a) showed experimentally for the benchmark tasks from previous International Planning Competitions (IPC) that saturated cost partitioning (SCP) (Seipp and Helmert 2014; 2018) is the cost partitioning algorithm of choice for PDB heuristics. Saturated cost partitioning works on an ordered sequence of heuristics. Iteratively, it gives each heuristic the minimum amount of costs that the heuristic needs to justify all its estimates and then uses the remaining costs for subsequent heuristics until all heuristics have been served this way.

Before we can compute a saturated cost partitioning over pattern database heuristics, we need to select a collection of patterns. The first domain-independent automated pattern selection algorithm is due to Edelkamp (2001). It partitions the state variables into patterns via best-fit bin packing. Edelkamp (2006) later used a genetic algorithm to search for a pattern collection that maximizes the average heuristic value of a zero-one cost partitioning over the PDB heuristics.

Haslum et al. (2007) proposed an algorithm that performs a hill-climbing search in the space of pattern collections (HC). HC evaluates a collection C by estimating the search effort of the canonical heuristic over C based on a model of IDA* runtime (Korf, Reid, and Edelkamp 2001).

Franco et al. (2017) presented the Complementary PDBs Creation (CPC) method, that combines bin packing and genetic algorithms to create a pattern collection minimizing the estimated search effort of an A∗ search (Lelis, Stern, and Sturtevant 2014).

Rovner, Sievers, and Helmert (2017) introduced a pattern selection algorithm (CEGAR) based on counterexample-guided abstraction refinement. The algorithm repeatedly computes patterns using the CEGAR principle: starting from
a random goal variable, it iteratively finds solutions in the corresponding projection and executes them in the original state space. Whenever a solution cannot be executed due to a violated precondition, it adds the missing precondition variable to the pattern.

Finally, Pommerening, Röger, and Helmert (2013) systematically generated all interesting patterns up to a given size $X$ (Sys-X). Experiments showed that cost-partitioned heuristics over Sys-2 and Sys-3 yield accurate estimates (Pommerening, Röger, and Helmert 2013; Seipp, Keller, and Helmert 2017a), but using all interesting patterns of larger sizes is usually infeasible.

We introduce Sys-SCP, a new pattern selection algorithm based on saturated cost partitioning that potentially considers all interesting patterns, but only selects useful ones. Sys-SCP builds multiple pattern sequences that together form the resulting pattern collection. For each sequence $\sigma$, it considers the interesting patterns in increasing order by size and adds a pattern $P$ to $\sigma$ if $P$ is not part of an earlier sequence and the saturated cost partitioning heuristic over $\sigma$ plus $P$ is more informative than the one over $\sigma$ alone.

**Background**

We consider optimal classical planning tasks in a SAS+ like notation (Bäckström and Nebel 1995) and represent a planning task $\Pi$ as a tuple $\langle V, O, s_0, s_*$ $\rangle$. Each variable $v$ in the finite set of variables $V$ has a finite domain $\text{dom}(v)$. A partial state $s$ is defined over a subset of variables $\text{vars}(s) \subseteq V$ and maps each $v \in \text{vars}(s)$ to a value in $\text{dom}(v)$, written as $s[v]$. We call the pair $\langle v, s[v] \rangle$ an atom and interchangeably treat partial states as mappings from variables to values or as sets of atoms. If $\text{vars}(s) = V$, we call $s$ a state. We write $S(\Pi)$ for the set of all states in $\Pi$.

Each operator $o$ in the finite set of operators $O$ has a precondition $\text{pre}(o)$ and an effect $\text{eff}(o)$, both of which are partial states, and a cost $\text{cost}(o) \in \mathbb{R}_+$. An operator $o$ is applicable in a state $s$ if $\text{pre}(o) \subseteq s$. Applying $o$ in $s$ leads into state $s' = s \circ o$ with $s'[v] = \text{eff}(o)[v]$ for all $v \in \text{vars}(\text{eff}(o))$ and $s'[v] = s[v]$ for all variables $v \in V \setminus \text{vars}(\text{eff}(o))$. The state $s_0$ is called the initial state and $s_*$ is a partial state, the goal.

Transition systems assign semantics to planning tasks.

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

A planning task $\Pi = \langle V, O, s_0, s_* \rangle$ induces a transition system $T$ with states $S(\Pi)$, labels $O$, transitions $\{ s \xrightarrow{a} s' | s \in S(\Pi), a \in O, \text{pre}(a) \subseteq s \}$, initial state $s_0$ and goal states $\{ s \in S(\Pi) | s_* \subseteq s \}$.

Separating transition systems from cost functions allows us to evaluate the same transition system under different cost functions, which is important for cost partitioning.

**Definition 2. Cost Functions.** A cost function for transition system $T$ is a function cost : $L(T) \rightarrow \mathbb{R} \cup \{-\infty, \infty\}$. A cost function cost is finite if $-\infty < \text{cost}(\ell) < \infty$ for all labels $\ell$. It is non-negative if $\text{cost}(\ell) \geq 0$ for all labels $\ell$.

We write $C(T)$ for the set of all cost functions for $T$ and $C_{\geq 0}(T)$ for the set of non-negative cost functions for $T$.

We call a cost function general when we want to emphasize that it might be negative and infinite. Note that we assume that the cost function of the planning task is finite and non-negative, but following Seipp and Helmert (2019) we allow general cost functions in cost partitionings.

**Definition 3. Weighted Transition Systems.** A weighted transition system is a pair $\langle T, \text{cost} \rangle$ where $T$ is a transition system and $\text{cost} \in C(T)$ is a cost function for $T$.

The cost of a path $\pi = \langle s^0 \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n \rangle$ in a weighted transition system $\langle T, \text{cost} \rangle$ is defined as $\text{cost}(\pi) = \sum_{i=1}^{n} \text{cost}(\ell_i)$. It is infinite if the sum contains both $+\infty$ and $-\infty$. If $s^n$ is a goal state, $\pi$ is called a goal path for $s^0$.

**Definition 4. Goal Distances and Optimal Paths.** The goal distance of a state $s \in S(T)$ in a weighted transition system $\langle T, \text{cost} \rangle$ is defined as $\inf_{\pi \in \Pi(s, T)} \text{cost}(\pi)$, where $\Pi(s, T)$ is the set of goal paths from $s \in T$. (The infimum of the empty set is $\infty$.) We write $h_T^*(\text{cost}(s))$ for the goal distance of $s$. If $h_T^*(\text{cost}(s), s) = \infty$, we call $s$ unsolvable.

A goal path $\pi$ from $s$ is optimal if $\text{cost}(\pi) = h_T^*(\text{cost}(s))$.

Optimal classical planning is the problem of finding an optimal goal path from $s_0$ or showing that $s_0$ is unsolvable.

We use heuristics to estimate goal distances (Pearl 1984).

**Definition 5. Heuristics.** A heuristic for a transition system $T$ is a function $h : C(T) \times S(T) \rightarrow \mathbb{R} \cup \{-\infty, \infty\}$. Heuristic $h$ is admissible if $h(\text{cost}(s), s) \leq h_T^*(\text{cost}(s), s)$ for all $\text{cost} \in C(T)$ and all $s \in S(T)$.

Cost partitioning makes adding heuristics admissible by distributing the costs of each operator among the heuristics.

**Definition 6. Cost Partitioning.** Let $T$ be a transition system. A cost partitioning for a cost function $\text{cost} \in C(T)$ is a tuple $\langle \text{cost}_1, \ldots, \text{cost}_n \rangle \in C(T)^n$ whose sum is bounded by cost : $\sum_{i=1}^{n} \text{cost}_i(\ell) \leq \text{cost}(\ell)$ for all $\ell \in L(T)$. A cost partitioning $\langle \text{cost}_1, \ldots, \text{cost}_n \rangle \in C(T)^n$ induces the cost-partitioned heuristic $h(\text{cost}, s) = \sum_{i=1}^{n} h_i(\text{cost}_i, s)$. If the sum contains $+\infty$ and $-\infty$, it evaluates to the leftmost infinite minimum.

One of the cost partitioning algorithms from the literature is saturated cost partitioning (Seipp and Helmert 2018). It is based on the insight that we can often reduce the amount of costs given to a heuristic without changing any heuristic estimates. Saturated cost functions formalize this idea.

**Definition 7. Saturated Cost Functions.** Consider a transition system $T$, a heuristic $h$ for $T$ and a cost function $\text{cost} \in C(T)$. A cost function $\text{scf} \in C(T)$ is saturated for $h$ and cost if

1. $\text{scf} \leq \text{cost}$ and
2. $h(\text{scf}, s) = h(\text{cost}, s)$ for all states $s \in S(T)$. 

A saturated cost function scf is minimal if there is no other saturated cost function scf for h and cost with scf(ℓ) ≤ scf(ℓ) for all labels ℓ ∈ L(T).

Whether we can efficiently compute a minimal saturated cost function depends on the type of heuristic. Seipp and Helmert (2014) showed that we can efficiently compute minimal saturated cost functions for explicitly-represented abstraction heuristics, including PDB heuristics.

Let T be a transition system, cost ∈ C_{≥0}(T) a non-negative cost function for T and h an abstraction heuristic for T with the abstract transition system T'. Then the minimum saturated cost function mscf for h and cost is defined as

\[ \text{mscf}(ℓ) = \sup_{a \rightarrow b \in T'(T')} (h^*_T(a, h^*_T(b)) \text{ for all } ℓ \in L(T). \]

The supremum of the empty set is −∞.

Consider a transition system T and a sequence of abstraction heuristics H = (h_1, ..., h_n) for T. For all 1 ≤ i ≤ n, saturate_i : C(T) → C(T) receives a cost function cost and returns the minimum saturated cost function for h_i and cost.

The saturated cost partitioning (cost_1, ..., cost_n) of a non-negative cost function cost ∈ C_{≥0}(T) over H is defined as:

\[ \text{rem}_0 = \text{cost} \]
\[ \text{cost}_i = \text{saturate}_i(\text{rem}_{i-1}) \text{ for all } 1 ≤ i ≤ n \]
\[ \text{rem}_i = \text{rem}_{i-1} - \text{cost}_i \text{ for all } 1 ≤ i ≤ n, \]

where the auxiliary cost functions rem_i represent the remaining costs after processing the first i heuristics in H.

We write h^\text{scp}_H for the saturated cost partitioning heuristic over the sequence of heuristics H. In this work, we compute saturated cost partitionings over pattern database heuristics.

A pattern for task II with variables V is a subset P ⊆ V. By syntactically removing all variables from PI that are not in P, we obtain the projected task II|P inducing the abstract transition system T_P. The PDB heuristic h^P for a pattern P is defined as h^P(cost, s) = h^\text{scp}_P(cost, s|P). For the pattern sequence \( P_1, ..., P_n \) we define h^\text{scp}_{P_1, ..., P_n} = h^\text{scp}_{P_1} \circ \cdots \circ h^\text{scp}_{P_n}.

One of the simplest pattern selection algorithms is to generate all patterns up to a given size X (Felner, Korf, and Hanan 2004) and we call this approach SYS-NAIVE-X. It is easy to see that for tasks with n variables, SYS-NAIVE-X generates \( \sum_{i=1}^{X} \binom{n}{i} \) patterns. Usually, many of these patterns do not add much information to a cost-partitioned heuristic over the patterns. Unfortunately, there is no efficiently computable test that allows us to discard such uninformative patterns. Even patterns without any goal variables can increase heuristic estimates in a general cost partitioning (Pommerening 2017).

However, in the setting where only non-negative cost functions are allowed in cost partitionings, there are efficiently computable criteria for deciding whether a pattern

Algorithm 1 SYS-SCP: Given a planning task with states \( S(T) \), cost function cost and interesting patterns Sys, select a subset \( C \subseteq Sys \).

1: function SYS-SCP(II)
2: \( C \leftarrow \emptyset \)
3: repeat
4: \( \sigma \leftarrow \emptyset \)
5: for each \( P \in \text{ORDER PATTERNS}(Sys) \) do
6: \( \text{if } P \in C \text{ then continue} \)
7: \( \text{if } P \text{Pattern Useful}(C, \sigma, P) \text{ then} \)
8: \( \sigma \leftarrow \sigma \oplus P \)
9: \( C \leftarrow C \cup \{ P \} \)
10: until \( \sigma = \emptyset \)
11: return \( C \)

12: function PatternUseful(C, \sigma, P)
13: \( \exists s \in S(T) : \)
14: \( h^\sigma_{scp}(cost, s) < h^\text{scp}_{P}(cost, s) < \infty \lor \max_{P' \in C} h^{P'}_{scp}(cost, s) < h^P(cost, s) = \infty \)

is interesting, i.e., whether it cannot be replaced by a set of smaller patterns that together yield the same heuristic estimates (Pommerening, Röger, and Helmert 2013).

The criteria are based on the causal graph CG(II) of a planning task II. CG(II) is a directed graph with a node for each variable in II. If there is an operator with a precondition on \( u \) and an effect on \( v \neq u \), CG(II) contains a precondition arc from \( u \) to \( v \). If an operator affects both \( u \) and \( v \), CG(II) contains a co-effect arc from \( u \) to \( v \) and from \( v \) to \( u \).

Definition 10. Interesting Patterns.
A pattern \( P \) is interesting if
1. CG(II|P) is weakly connected, and
2. CG(II|P) contains a directed path via precondition arcs from each node to some goal variable node.

The systematic pattern generation method SYS-X generates all interesting patterns up to size X. We let SYS denote the set of all interesting patterns for a given task. On IPC benchmark tasks, SYS-X often generates much fewer patterns than SYS-NAIVE-X for the same size limit X. Still, it is usually infeasible to compute all SYS-X patterns and the corresponding projections for X > 3 within reasonable amounts of time and memory. Also, we hypothesize that even when considering only interesting patterns, usually only a small percentage of the systematic patterns up to size 3 contribute much information to the resulting heuristic.

For these two reasons we propose a new pattern selection algorithm that potentially considers all interesting patterns, but only selects the ones that it deems useful.

Sys-SCP Pattern Selection Algorithm.
Our new pattern selection algorithm repeatedly creates a new empty pattern sequence \( \sigma \) and only appends those interesting patterns to \( \sigma \) that increase any heuristic values of a saturated cost partitioning heuristic computed over \( \sigma \).

Algorithm 1 shows pseudo-code for the procedure, which we call SYS-SCP. It starts with an empty pattern collection.
Consider a planning task $\Pi$ with non-negative cost function $\sigma$ and induced transition system $T$. Let $P$ be a pattern for $\Pi$ and $\sigma$ be a (possibly empty) sequence of patterns $(P_1, \ldots, P_n)$ for $\Pi$. Finally, let $\text{rem}$ be the remaining cost function after computing $h_{SCP}^\Pi$ for cost.

$$h_{SCP,\sigma}^\Pi (\text{cost}, s) < h_{SCP,\sigma\oplus P}^\Pi (\text{cost}, s) < \infty$$

$$\iff 0 < h_{SCP}^P (\text{rem}, s|P) < \infty$$

Proof.

1. $$h_{SCP}^\Pi (\text{cost}, s) < h_{SCP,\sigma\oplus P}^\Pi (\text{cost}, s) < \infty$$
2. $$\forall i \in [1, n] h_{SCP}^{P_i} (\text{cost}, s) < h_{SCP,\sigma\oplus P}^\Pi (\text{cost}, s)$$
3. $$\forall i \in [1, n] h_{SCP}^{P_i} (\text{cost}, s) < h_{SCP}^{P_i} (\text{cost}, s) + h_{SCP}^P (\text{rem}, s) < \infty$$
4. $$0 < h_{SCP}^{P_i} (\text{rem}, s) < \infty$$
5. $$0 < h_{SCP}^{P_i} (\text{rem}, s|P) < \infty$$

Step 1 substitutes $(P_1, \ldots, P_n)$ for $\sigma$ and Step 2 uses the definition of saturated cost partitioning heuristics. For Step 3 we need to show that $x = \sum_{i=1}^n h_{SCP}^{P_i} (\text{cost}, s)$ is finite.

The inequality states $x < \infty$. We now show $x > -\infty$. Requirement 1 for saturated cost functions (Definition 7) implies that $\text{rem}_0$ is non-negative for all $1 \leq i \leq n$. Since $\text{cost}$ and therefore $\text{rem}_0$ are also non-negative, requirement 2 for saturated cost functions ensures that $h_{SCP}^{P_i} (\text{cost}, s) \geq 0$ for all $1 \leq i \leq n$ and thus $x \geq 0 > -\infty$.

The final step (4) uses the definition of PDB heuristics.

We now show how to check on the level of projections whether a pattern detects new unsolvable states. It is easy to see that each abstract state in a projection corresponds to a partial state in the original task. If an abstract state is unsolvable in a projection, we call the corresponding partial state a dead end. Since projections preserve all paths, all states in the original task consistent with a dead end are unsolvable. Moreover, all partial states consistent with a dead end are also dead ends.

Definition 11. Dead end detection with PDB heuristics.

Let $P$ be a pattern for a planning task $\Pi$ with cost function $\text{cost}$ and let $s$ be a partial state in $\Pi$.

$P$ detects that $s$ is a dead end iff there is an unsolvable abstract state $s' \in S(T_P)$ that is consistent with $s$, i.e., $\exists s' \in S(T_P) : h_{SCP}^{P_s} (\text{cost}, s') = \infty \land s' \subseteq s$.

We let the predicate $\text{dead}(P, s)$ indicate whether $P$ detects that $s$ is a dead end.

Lemma 1 and Definition 11 let us decide whether a pattern is useful just by inspecting the projections.


Consider a planning task $\Pi$ with non-negative cost function $\sigma$ and induced transition system $T$. Let $P$ be a single pattern, $\sigma$ be a (possibly empty) sequence of patterns and $\Pi$ be a collection of patterns for $\Pi$. Finally, let $\text{rem}$ be the remaining cost function after computing $h_{SCP,\sigma}^\Pi$ for cost.

$$\exists s \in S(T) : h_{SCP,\sigma}^\Pi (\text{cost}, s) < h_{SCP,\sigma\oplus P}^\Pi (\text{cost}, s) < \infty$$

$$\forall P \in C : h_{SCP,\sigma\oplus P}^\Pi (\text{cost}, s) = \infty$$

$$\iff \exists s \in S(T_P) : h_{SCP}^P (\text{rem}, s') < \infty$$

Proof. Follows directly from Lemma 1, Definition 11 and the fact that projections are induced abstractions, where all abstract states have a nonempty preimage.

We use Theorem 1 in our Sys-SCP implementation by keeping track of the cost function $\text{rem}$, i.e., the costs that remain after computing $h_{SCP}^\Pi$, and the set of dead ends recognized by the pattern collection $C$. We select a pattern $P$ if there are any goal distances $d$ with $0 < d < \infty$ in $T_P$ under $\text{rem}$ or if $P$ detects a dead end that is not subsumed by any dead end detected by the patterns in $C$.

Theorem 1 also removes the need to compute $h_{SCP,\sigma\oplus P}^\Pi$ from scratch for every pattern $P$. This is important since we want to decide whether or not to add $P$ quickly and this operation should not become slower the more patterns are already part of $\sigma$.

Time Limits

The number of interesting patterns is huge for all but the smallest planning tasks and we cannot hope to generate or even evaluate them all in the inner loop of Sys-SCP. Therefore, we allow imposing time limits on the inner ($T_x$) and outer loop ($T_y$) of the algorithm. If $T_x < T_y$, Sys-SCP can return to evaluating smaller patterns multiple times instead of spending almost all of its time evaluating large patterns.

Ordering Patterns

Seipp, Keller, and Helmert (2017b) showed that the order in which saturated cost partitioning considers the component heuristics has a strong influence on the quality of the resulting heuristic. Choosing a good order is even more important for Sys-SCP, since it usually only sees a subset of interesting patterns within the allotted time.

Since the number of states in a projection is exponential in the pattern size, so is the time for computing PatternUseful. We hypothesize that it is more beneficial to consider many smaller patterns than few large patterns.
and therefore let the ORDERPATTERNS function sort the interesting patterns by size in increasing order.

This leaves the question how to sort patterns of the same size. We propose four methods for making this decision. The first one (random) simply orders patterns of the same size randomly. The remaining three assign a key to each pattern, allowing us to sort by key in increasing or decreasing order.

Causal Graph The first ordering method is based on the insight that it is often more important to have accurate heuristic estimates near the goal states rather than elsewhere in the state space (e.g., Holte et al. 2006; Torralba, Linares López, and Borrajo 2018). We therefore want to focus on patterns containing goal variables or variables that are closely connected to goal variables. To quantify “goal-connectedness” we use an approximate topological ordering \( \pi \) of the causal graph \( \mathcal{CG}(\Pi) \). We let the function \( \text{cg} : \mathcal{V} \to \mathbb{N}_0^+ \) assign each variable \( v \in \mathcal{V} \) to its index in \( \pi \). For a given pattern \( P \), the cg ordering method returns the key \( (\text{cg}(v_1), \ldots, \text{cg}(v_n)) \), where \( v_i \in P \) and \( \text{cg}(v_i) < \text{cg}(v_j) \) for all \( 1 \leq i < j \leq n \). Since the keys are unique, they define a total order. Sorting the patterns by \( \text{cg} \) in decreasing order (\( \text{cg-down} \)), yields the desired order which starts with “goal-connected” patterns.

States in Projection Given a pattern \( P \) the ordering method states returns the key \( |S(\Pi \rho)| \), i.e., the number of states in the projection to \( P \). We use \( \text{cg-down} \) to break ties.

Active Operators Given a pattern \( P \), the \( \text{ops} \) ordering method returns the number of operators that affect a variable in \( P \). We break ties with \( \text{cg-down} \).

Experiments

We implemented the Sys-SCP pattern selection algorithm in the Fast Downward planning system (Helmert 2006) and make the code publicly available.\(^1\) We run experiments with the Downward Lab toolkit (Seipp et al. 2017). Our benchmark set\(^2\) consists of all 1827 SAS\(^+\) tasks from the optimization tracks of the IPCs from 1998 to 2018. The tasks belong to 48 different domains, some of which were used in several IPCs. We limit time by 30 minutes and memory by 3.5 GiB. All experimental data is published online.\(^3\)

To fairly compare the quality of different pattern collections, we use the same cost partitioning algorithm for all collections. Saturated cost partitioning is the obvious choice for the evaluation since experiments showed that it is preferable to all other cost partitioning algorithms for HC, SYS-2 and CPC patterns in almost all evaluated benchmark domains (Seipp, Keller, and Helmert 2017a; Rovner, Sievers, and Helmert 2019).

\(^1\)Code: Zenodo link removed for anonymity
\(^2\)Benchmarks: Zenodo link removed for anonymity
\(^3\)Experimental data: Zenodo link removed for anonymity

Table 1: Number of solved tasks by SYS-SCP using different time limits for the inner loop (x axis) and outer loop (y axis).

<table>
<thead>
<tr>
<th>Coverage</th>
<th>1s</th>
<th>10s</th>
<th>100s</th>
<th>( \infty )</th>
</tr>
</thead>
<tbody>
<tr>
<td>30s</td>
<td>1129</td>
<td>1133</td>
<td>1071</td>
<td>1071</td>
</tr>
<tr>
<td>100s</td>
<td>1093</td>
<td>1153</td>
<td>1070</td>
<td>1070</td>
</tr>
<tr>
<td>900s</td>
<td>620</td>
<td>1054</td>
<td>1141</td>
<td>1010</td>
</tr>
<tr>
<td>( \infty )</td>
<td>412</td>
<td>264</td>
<td>248</td>
<td>211</td>
</tr>
</tbody>
</table>

Diverse Saturated Cost Partitioning Heuristics For a given pattern collection \( C \), we compute diverse saturated cost partitioning heuristics using the diversification procedure by Seipp, Keller, and Helmert (2017b); we start with an empty family of saturated cost partitioning heuristics \( \mathcal{F} \) and a set \( \mathcal{S} \) of 1000 sample states obtained with random walks (Haslum et al. 2007). Then we iteratively sample a new state \( s \) and compute a greedy order \( \omega \) of \( C \) that works well for \( s \) (Seipp 2017). If \( h^{\omega}_{\text{SCP}} \) has a higher heuristic estimate for any state \( s' \in \mathcal{S} \) than all heuristics in \( \mathcal{F} \), we add \( h^{\omega}_{\text{SCP}} \) to \( \mathcal{F} \). We stop this diversification procedure after 200 seconds and then perform an A* search using the maximum over the heuristics in \( \mathcal{F} \).

Before we compare SYS-SCP to other pattern selection algorithms, we evaluate the effects of changing its parameters in three ablation studies. We use at most 2M states per PDB and 20M states in the PDB collection for all SYS-SCP runs.

Time Limits

Table 1 shows that a time limit for the outer loop is more important than one for the inner loop, but for maximum coverage we need both limits. The combination that solves the highest number of tasks is 10s for the inner and 100s for the outer loop. We use these values in all remaining experiments.

Pattern Orders

Table 2 compares the different methods for ordering patterns of the same size. Using a random order leads to the worst results. For the ops and states ordering methods, roughly the same number of domains profit from an increasing and from a decreasing order by key. The only ordering method for which the domains clearly prefer one direction is cg: cg-down solves more tasks than cg-up in 12 domains, while the opposite is the case in only 4 domains. Since cg-down also has the highest overall coverage, we use it in all other experiments.

Using Pattern Sequences for Diversification

Instead of discarding the computed pattern sequences when SYS-SCP finishes, we can turn each pattern sequence \( \sigma \) into a full pattern order by appending all SYS-SCP patterns missing from \( \sigma \) to \( \sigma \) in an arbitrary order and use it in the diversification procedure.

Feeding the diversification exclusively with such orders leads to solving 1118 tasks, while using only greedy orders
Table 2: Per-domain coverage comparison of different orders for patterns of the same size. The entry in row \( r \) and column \( c \) shows the number of domains in which order \( r \) solves more tasks than order \( c \). For each order pair we highlight the maximum of the entries \((r, c)\) and \((c, r)\) in bold. Right: Total number of solved tasks. The results for \( \text{random} \) are averaged over 10 runs.

<table>
<thead>
<tr>
<th>Max pattern size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{SYS-Naive} )</td>
<td>840</td>
<td>932</td>
<td>903</td>
<td>743</td>
<td>568</td>
</tr>
<tr>
<td>( \text{SYS-Naive-LIM} )</td>
<td>839</td>
<td>958</td>
<td>992</td>
<td>902</td>
<td>864</td>
</tr>
<tr>
<td>( \text{SYS} )</td>
<td>840</td>
<td>983</td>
<td>1049</td>
<td>911</td>
<td>724</td>
</tr>
<tr>
<td>( \text{SYS-LIM} )</td>
<td>839</td>
<td>983</td>
<td>1081</td>
<td>1029</td>
<td>1002</td>
</tr>
</tbody>
</table>

Table 3: Number of solved tasks for naive (\( \text{SYS-Naive} \)) and interesting patterns (\( \text{SYS} \)). We evaluate both versions without and with time and memory limits and using different maximum pattern sizes.

<table>
<thead>
<tr>
<th>Coverage</th>
<th>( \text{random} )</th>
<th>( \text{cg-up} )</th>
<th>( \text{ops-down} )</th>
<th>( \text{states-up} )</th>
<th>( \text{states-down} )</th>
<th>( \text{ops-up} )</th>
<th>( \text{cg-down} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{random} )</td>
<td>–</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>( \text{cg-up} )</td>
<td>11</td>
<td>–</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>( \text{ops-down} )</td>
<td>11</td>
<td>6</td>
<td>–</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>( \text{states-up} )</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>–</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>( \text{states-down} )</td>
<td>12</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>–</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>( \text{ops-up} )</td>
<td>14</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>–</td>
<td>7</td>
</tr>
<tr>
<td>( \text{cg-down} )</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4: Per-domain coverage comparison of pattern selection algorithms. For an explanation of the data see the caption of Table 2.

Comparison to IPC Planners

In our final experiment, we evaluate whether Scorpion (Seipp 2018), one of the strongest planners in IPC 2018, benefits from using SYSCP patterns. Scorpion computes diverse saturated cost partitioning heuristics over HC and SYSCP. SYSCP has the highest total coverage of 1153 tasks, solving 49 more tasks than the strongest contender. This is a considerable improvement in the setting of optimal classical planning, where task difficulty tends to scale exponentially.

Conclusion

We introduced a new pattern selection algorithm based on saturated cost partitioning and showed that it outperforms all other pattern selection algorithms from the literature. The
The algorithm selects a pattern if it is useful for any state in the state space. In future work, we would like to evaluate whether it is beneficial to restrict this criterion to a subset of states, such as all reachable states or a set of sample states.

References


Table 5: Comparison of IPC 2018 planners and Scorpion variants.

<table>
<thead>
<tr>
<th>Scorpion</th>
<th>Complementary1</th>
<th>Complementary2</th>
<th>Planning-PDBs</th>
<th>SYS-SCP</th>
<th>SYS-SCP+COMB</th>
<th>SYS-SCP+CART</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage</td>
<td>1027</td>
<td>1097</td>
<td>1096</td>
<td>1199</td>
<td>1200</td>
<td>1242</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scorpion</th>
<th>Complementary1</th>
<th>Planning-PDBs</th>
<th>SYS-SCP</th>
<th>SYS-SCP+CART</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage</td>
<td>1027</td>
<td>1096</td>
<td>1199</td>
<td>1200</td>
</tr>
</tbody>
</table>

### Table 5: Comparison of IPC 2018 planners and Scorpion variants.

The algorithm selects a pattern if it is useful for any state in the state space. In future work, we would like to evaluate whether it is beneficial to restrict this criterion to a subset of states, such as all reachable states or a set of sample states.
