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.

1 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 (2019) 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 σ, it considers the interesting patterns in increasing order by size and adds a pattern P to σ if P is not part of an earlier sequence and the saturated cost partitioning heuristic over σ plus P is more informative than the one over σ alone.

2 Background

We consider optimal classical planning tasks in a SAS$^+$-like notation (Bäckström and Nebel 1995) and represent a planning task Π as a tuple $\langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$. Each variable v in the finite set of variables $\mathcal{V}$ has a finite domain $\mathit{dom}(v)$. A *partial state* s is defined over a subset of variables $\mathit{vars}(s) \subseteq \mathcal{V}$ and maps each $v \in \mathit{vars}(s)$ to a value in $\mathit{dom}(v)$, 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 $\mathit{vars}(s) = \mathcal{V}$, we call s a *state*. We write $S(\Pi)$ for the set of all states in Π.

Each operator o in the finite set of operators $\mathcal{O}$ has a *precondition* $\mathit{pre}(o)$ and an *effect* $\mathit{eff}(o)$, both of which are partial states, and a cost $\mathit{cost}(o) \in \mathbb{R}_0^+$. An operator o is applicable in a state s if $\mathit{pre}(o) \subseteq s$. Applying o in s leads into state $s' = s[\![o]\!]$ with $s'[v] = \mathit{eff}(o)[v]$ for all $v \in \mathit{vars}(\mathit{eff}(o))$ and $s'[v] = s[v]$ for all variables $v \in \mathcal{V} \setminus \mathit{vars}(\mathit{eff}(o))$. The state s_0 is called the *initial state* and $s_\star$ is a partial state, the *goal*.

Transition systems assign semantics to planning tasks.

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

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ induces a transition system $\mathcal{T}$ with states $S(\Pi)$, labels $\mathcal{O}$, transitions $\{s \xrightarrow{o} s[\![o]\!] \mid s \in S(\Pi), o \in \mathcal{O}, \mathit{pre}(o) \subseteq s\}$, initial state s_0 and goal states $\{s \in S(\Pi) \mid s_\star \subseteq s\}$.

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 $\mathcal{T}$ is a function cost : $L(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$. A cost function cost is* finite *if $-\infty < \mathit{cost}(\ell) < \infty$ for all labels ℓ. It is* non-negative *if $\mathit{cost}(\ell) \geq 0$ for all labels ℓ.*

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

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 $\langle \mathcal{T}, \mathit{cost} \rangle$ where $\mathcal{T}$ is a transition system and cost $\in \mathcal{C}(\mathcal{T})$ is a cost function for $\mathcal{T}$.*

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

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

A goal path π from s is optimal *if $\mathit{cost}(\pi) = h^*_{\mathcal{T}}(\mathit{cost}, s)$.*

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 $\mathcal{T}$ is a function $h : \mathcal{C}(\mathcal{T}) \times S(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$.*

Heuristic h is admissible *if $h(\mathit{cost}, s) \leq h^*_{\mathcal{T}}(\mathit{cost}, s)$ for all cost $\in \mathcal{C}(\mathcal{T})$ and all $s \in S(\mathcal{T})$.*

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

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

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 $\mathcal{T}$, a heuristic h for $\mathcal{T}$ and a cost function cost $\in \mathcal{C}(\mathcal{T})$. A cost function scf $\in \mathcal{C}(\mathcal{T})$ is saturated *for h and cost if*

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

A saturated cost function scf is minimal *if there is no other saturated cost function scf′ for h and cost with scf$(\ell) \leq$ scf′(ℓ) for all labels $\ell \in L(\mathcal{T})$.*

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.

Definition 8. ***Minimum Saturated Cost Function for Abstraction Heuristics.***
Let $\mathcal{T}$ be a transition system, cost $\in \mathcal{C}_{\geq 0}(\mathcal{T})$ a non-negative cost function for $\mathcal{T}$ and h an abstraction heuristic for $\mathcal{T}$ with the abstract transition system $\mathcal{T}'$. Then the minimum saturated cost function *mscf for h and cost is defined as*

$$\mathit{mscf}(\ell) = \sup_{\substack{a \xrightarrow{\ell} b \in T(\mathcal{T}'),\\ h^*_{\mathcal{T}'}(\mathit{cost}, b) < \infty}} (h^*_{\mathcal{T}'}(\mathit{cost}, a) - h^*_{\mathcal{T}'}(\mathit{cost}, b))$$

for all $\ell \in L(\mathcal{T})$. The supremum of the empty set is $-\infty$.

Definition 9. ***Saturated Cost Partitioning.***
Consider a transition system $\mathcal{T}$ and a sequence of abstraction heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ for $\mathcal{T}$. For all $1 \leq i \leq n$, saturate$_i : \mathcal{C}(\mathcal{T}) \to \mathcal{C}(\mathcal{T})$ receives a cost function cost′ and returns the minimum saturated cost function for h_i and cost′.

The saturated cost partitioning $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle$ *of a non-negative cost function cost $\in \mathcal{C}_{\geq 0}(\mathcal{T})$ over $\mathcal{H}$ is defined as:*

$$\begin{aligned} \mathit{rem}_0 &= \mathit{cost} \\ \mathit{cost}_i &= \mathit{saturate}_i(\mathit{rem}_{i-1}) && \text{for all } 1 \leq i \leq n \\ \mathit{rem}_i &= \mathit{rem}_{i-1} - \mathit{cost}_i && \text{for all } 1 \leq i \leq n, \end{aligned}$$

where the auxiliary cost functions rem$_i$ represent the remaining costs after processing the first i heuristics in $\mathcal{H}$.

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

A *pattern* for task Π with variables $\mathcal{V}$ is a subset $P \subseteq \mathcal{V}$. By syntactically removing all variables from Π that are not in P, we obtain the *projected* task $\Pi|_P$ inducing the abstract transition system $\mathcal{T}_P$. The PDB heuristic h^P for a pattern P is defined as $h^P(\mathit{cost}, s) = h^*_{\mathcal{T}_P}(\mathit{cost}, s|_P)$. For the pattern sequence $\langle P_1, \ldots, P_n \rangle$ we define $h^{\text{SCP}}_{\langle P_1, \ldots, P_n \rangle} = h^{\text{SCP}}_{\langle h^{P_1}, \ldots, h^{P_n} \rangle}$.

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 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).

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

```
function SYS-SCP(Π)
    C ← ∅
    repeat
        σ ← ⟨⟩
        for each P ∈ ORDERPATTERNS(SYS) do
            if P ∈ C then continue
            if PATTERNUSEFUL(C, σ, P) then
                σ ← σ ⊕ P
                C ← C ∪ {P}
    until σ = ⟨⟩
    return C

function PATTERNUSEFUL(C, σ, P)
    return ∃s ∈ S(T) :
        h^SCP_σ(cost, s) < h^SCP_{σ⊕P}(cost, s) < ∞ ∨
        max_{P′∈C} h^{P′}(cost, s) < h^P(cost, s) = ∞
```

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

Definition 10. ***Interesting Patterns.***
A pattern P is interesting *if*

1. *$CG(\Pi|_P)$ is weakly connected, and*
2. *$CG(\Pi|_P)$ 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.

3 Sys-SCP Pattern Selection Algorithm

Our new pattern selection algorithm repeatedly creates a new empty pattern sequence σ and only appends those interesting patterns to σ that increase any heuristic values of a saturated cost partitioning heuristic computed over σ.

Algorithm 1 shows pseudo-code for the procedure, which we call SYS-SCP. It starts with an empty pattern collection

C. In each iteration of the outer loop, SYS-SCP creates a new empty pattern sequence σ and then loops over the interesting patterns $P \in$ SYS in the order chosen by ORDERPATTERNS (see Section 3.2). SYS-SCP appends a pattern P to σ and includes it in C if there is a state s for which the saturated cost partitioning over σ extended by P has a higher finite heuristic value than the one over σ alone or if P detects that s is unsolvable but C fails to do so. Once an iteration adds no new patterns, the algorithm stops and returns C.

The most important component of the SYS-SCP algorithm is the PATTERNUSEFUL function that decides whether to select a pattern P. The function enumerates all states $s \in S(\Pi)$, which is obviously infeasible for all but the smallest tasks Π. Fortunately, we can efficiently compute an equivalent test in the projections. To make the first part of PATTERNUSEFUL efficient we use the following lemma.

Lemma 1.
Consider a planning task Π with non-negative cost function cost and induced transition system $\mathcal{T}$. Let $s \in S(\mathcal{T})$ be a state, P be a pattern for Π and σ be a (possibly empty) sequence of patterns $\langle P_1, \dots, P_n \rangle$ for Π. Finally, let rem be the remaining cost function after computing h^{SCP}_σ for cost.

$$h^{SCP}_\sigma(cost, s) < h^{SCP}_{\sigma \oplus P}(cost, s) < \infty$$
$$\Leftrightarrow\ 0 < h^*_{\mathcal{T}_P}(rem, s|_P) < \infty$$

Proof.

$$h^{\text{SCP}}_\sigma(cost, s) < h^{\text{SCP}}_{\sigma \oplus P}(cost, s) < \infty$$
$$\overset{(1)}{\Leftrightarrow} h^{\text{SCP}}_{\langle P_1, \dots, P_n \rangle}(cost, s) < h^{\text{SCP}}_{\langle P_1, \dots, P_n, P \rangle}(cost, s) < \infty$$
$$\overset{(2)}{\Leftrightarrow} \sum_{i=1}^{n} h^{P_i}(cost_i, s) < \sum_{i=1}^{n} h^{P_i}(cost_i, s) + h^P(rem, s) < \infty$$
$$\overset{(3)}{\Leftrightarrow} 0 < h^P(rem, s) < \infty \overset{(4)}{\Leftrightarrow} 0 < h^*_{\mathcal{T}_P}(rem, s|_P) < \infty$$

Step 1 substitutes $\langle P_1, \dots, P_n \rangle$ for σ and Step 2 uses the definition of saturated cost partitioning heuristics. For Step 3 we need to show that $x = \sum_{i=1}^{n} h^{P_i}(cost_i, s)$ is finite.

The inequality states $x < \infty$. We now show $x > -\infty$. Requirement 1 for saturated cost functions (Definition 7) implies that rem_i is non-negative for all $1 \leq i \leq n$. Since *cost* and therefore rem_0 are also non-negative, requirement 2 for saturated cost functions ensures that $h^{P_i}(cost_i, s) \geq 0$ 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 Π with cost function cost and let s be a partial state in Π.

P detects that s is a dead end iff there is an unsolvable abstract state $s' \in S(\mathcal{T}_P)$ that is consistent with s, i.e., $\exists s' \in S(\mathcal{T}_P) : h^*_{\mathcal{T}_P}(cost, s') = \infty \wedge s' \subseteq s$.

We let the predicate $dead(P, s)$ 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.

Theorem 1. ***Computing* PATTERNUSEFUL *on Projections.***
Consider a planning task Π with non-negative cost function cost and induced transition system $\mathcal{T}$. Let P be a single pattern, σ be a (possibly empty) sequence of patterns and C be a collection of patterns for Π. Finally, let rem be the remaining cost function after computing h^{SCP}_σ for cost.

$$\exists s \in S(\mathcal{T}) : h^{SCP}_\sigma(cost, s) < h^{SCP}_{\sigma \oplus P}(cost, s) < \infty$$
$$\vee \max_{P' \in C} h^{P'}(cost, s) < h^P(cost, s) = \infty$$
$$\Leftrightarrow \exists s' \in S(\mathcal{T}_P) : 0 < h^*_{\mathcal{T}_P}(rem, s') < \infty$$
$$\vee((\forall P' \in C : \neg dead(P', s')) \wedge dead(P, s'))$$

Proof. Follows directly from Lemma 1, Definition 11 and the fact that projections are induced abstractions, where all abstract states have a nonempty preimage, i.e., for each abstract state s' there is at least one concrete state s which is projected to s'. □

We use Theorem 1 in our SYS-SCP implementation by keeping track of the cost function *rem*, i.e., the costs that remain after computing h^{SCP}_σ, 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 $\mathcal{T}_P$ under *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 $h^{\text{SCP}}_{\sigma \oplus P}$ 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 σ.

3.1 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.

3.2 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 often 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 $\prec$ of the causal graph $CG(\Pi)$. We let the function $cg : \mathcal{V} \to \mathbb{N}_0^+$ assign each variable $v \in \mathcal{V}$ to its index in $\prec$. For a given pattern P, the *cg* ordering method returns the key $\langle cg(v_1), \dots, cg(v_n) \rangle$, where $v_i \in P$ and $cg(v_i) < cg(v_j)$ for all $1 \leq i < j \leq n$. Since the keys are unique, they define a total order. Sorting the patterns by *cg* in decreasing order (*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 $|S(\Pi|_P)|$, i.e., the number of states in the projection to P. We use *cg-down* to break ties.

Active Operators Given a pattern P, the *ops* ordering method returns the number of operators that affect a variable in P. We break ties with *cg-down*.

4 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).

Coverage	1s	10s	100s	∞
30s	1129	**1133**	1071	1071
100s	1093	**1153**	1070	1070
900s	620	1054	**1141**	1010
∞	**412**	264	248	211

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

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 $\hat{S}$ 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 ω of C that works well for s (Seipp 2017). If h^{SCP}_{ω} has a higher heuristic estimate for any state $s' \in \hat{S}$ than all heuristics in $\mathcal{F}$, we add h^{SCP}_{ω} 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.

4.1 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.

4.2 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.

4.3 Using Pattern Sequences for Diversification

Instead of discarding the computed pattern sequences when SYS-SCP finishes, we can turn each pattern sequence σ into

[1] Code: Zenodo link removed for anonymity

[2] Benchmarks: Zenodo link removed for anonymity

[3] Experimental data: Zenodo link removed for anonymity

	random	cg-up	ops-down	states-up	states-down	ops-up	cg-down	Coverage
random	–	9	7	8	6	8	2	1134.5
cg-up	**11**	–	5	5	6	4	4	1135.0
ops-down	**11**	**6**	–	**6**	2	7	2	1136.0
states-up	**12**	**8**	**6**	–	4	4	2	1140.0
states-down	**12**	**9**	**5**	**6**	–	7	1	1145.0
ops-up	**14**	**10**	**9**	**10**	**8**	–	7	**1153.0**
cg-down	**16**	**12**	**7**	**8**	**5**	**8**	–	**1153.0**

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 *random* are averaged over 10 runs.

Max pattern size	1	2	3	4	5
SYS-NAIVE	840	**932**	903	743	568
SYS-NAIVE-LIM	839	958	**992**	902	864
SYS	840	983	**1049**	911	724
SYS-LIM	839	983	**1081**	1029	1002

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

a full pattern order by appending all SYS-SCP patterns missing from σ to σ 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 for sample states (Seipp 2017) solves 1132 tasks. We obtain the best results by diversifying both types of orders, solving 1153 tasks, and we use this variant in all other experiments.

4.4 Systematic Patterns With Limits

In the next experiment, we evaluate the obvious baseline for SYS-SCP: selecting all (interesting) patterns up to a fixed size. Table 3 holds coverage results of SYS-NAIVE-X and SYS-X for $1 \leq X \leq 5$. We also include variants (*-LIM) that use at most 100 seconds, no more than 2M states in each projection and at most 20M states per collection. For the *-LIM variants, we sort the patterns in the *cg-down* order.

The results show that interesting patterns are always preferable to naive patterns, both with and without limits, which is why we only consider interesting patterns in SYS-SCP. Imposing limits is not important for SYS-1 and SYS-2, but leads to solving many more tasks for $X \geq 3$. Overall, SYS-3-LIM has the highest total coverage.

	HC	SYS-3-LIM	CPC	CEGAR	SYS-SCP	Coverage
HC	–	9	10	6	5	965
SYS-3-LIM	**19**	–	12	8	1	1081
CPC	**20**	**19**	–	11	9	1061
CEGAR	**22**	**17**	**17**	–	5	1104
SYS-SCP	**26**	**25**	**21**	**21**	–	**1153**

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

4.5 Comparison of Pattern Selection Algorithms

In Table 4 we compare SYS-SCP to the strongest pattern selection algorithms from the literature: HC, SYS-3-LIM, CPC and CEGAR. (See Table 5 for per-domain coverage results.) We run each algorithm with its preferred parameter values, which implies using at most 900s for HC and CPC and 100s for the other algorithms.

SYS-SCP outperforms even the strongest variant of the SYS-* algorithms from Section 4.4: it solves more tasks than SYS-3-LIM in 25 domains, while the opposite is the case in only one domain. SYS-SCP is also preferable to all other pattern selection algorithms in per-domain comparisons and it 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.

4.6 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 SYS-SCP patterns. Scorpion computes diverse saturated cost partitioning heuristics over HC and SYS-2 PDB heuristics and Cartesian abstraction heuristics (CART) (Seipp and Helmert 2018). We abbreviate this combination with COMB=HC+SYS-2+CART. In Table 6 we compare the original Scorpion planner, three Scorpion variants that use different sets of heuristics and the top three non-portfolio planners from IPC 2018, Complementary1 (Franco et al. 2018), Complementary2 (Franco et al. 2017) and Planning-PDBs (Moraru et al. 2018), which use symbolic PDBs. (Table 5 holds per-domain coverage results for these planners.) In contrast to the configurations we evaluated above, the three planners based on symbolic PDBs and the four Scorpion variants prune irrelevant operators in a preprocessing step (Alcázar and Torralba 2015).

The results show that all Scorpion variants outperform Complementary1/2 and Planning-PDBs in per-domain and total coverage comparisons. We also see that Scorpion benefits from using the SYS-SCP PDBs instead of the COMB heuristics in many domains, even if the total coverage remains roughly the same. Using the union of both sets

	h^{SCP} over different PDB heuristics					Symbolic PDBs			Scorpion			
Coverage	HC	SYS-3-LIM	CPC	CEGAR	SYS-SCP	Comp1	Comp2	PPDBs	COMB	SYS-SCP	SYS-SCP +COMB	SYS-SCP +CART
agricola (20)	0	0	0	3	0	**6**	**6**	**6**	1	3	0	1
airport (50)	38	23	31	37	32	21	25	25	39	**46**	45	45
barman (34)	4	4	**11**	4	**11**	**11**	**11**	**11**	4	**11**	**11**	**11**
blocks (35)	28	28	28	28	28	**30**	**30**	**30**	28	28	28	28
childsnack (20)	0	0	0	0	0	0	1	**4**	0	0	0	0
data-network (20)	12	11	13	13	13	12	13	13	**14**	13	**14**	**14**
depot (22)	11	13	9	11	13	7	7	7	14	14	**15**	14
driverlog (20)	13	**15**	14	**15**	**15**	13	14	14	**15**	**15**	**15**	**15**
elevators (50)	41	43	**44**	**44**	41	37	37	37	**44**	41	**44**	**44**
floortile (40)	6	11	15	11	13	**34**	**34**	**34**	16	21	21	21
freecell (80)	21	25	27	27	30	22	25	27	**68**	30	66	67
ged (20)	**19**	**19**	**19**	**19**	**19**	16	**19**	**19**	**19**	**19**	**19**	**19**
grid (5)	**3**	**3**	**3**	**3**	**3**	2	2	2	**3**	**3**	**3**	**3**
gripper (20)	8	8	7	8	8	**20**	**20**	**20**	8	8	8	8
hiking (20)	13	14	**19**	15	15	15	18	17	14	15	15	15
logistics (63)	30	33	30	34	35	26	28	28	35	**36**	34	**36**
miconic (150)	72	143	121	143	**144**	105	105	105	143	**144**	**144**	143
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	24	33	23	30	34	19	22	22	30	34	34	**35**
mystery (30)	17	**19**	16	**19**	**19**	13	15	16	**19**	**19**	**19**	**19**
nomystery (20)	**20**	**20**	**20**	**20**	**20**	12	19	18	**20**	**20**	**20**	**20**
openstacks (100)	49	51	49	49	53	**74**	**74**	**74**	51	53	53	53
organic (20)	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**
organic-split (20)	10	10	10	10	10	12	**13**	**13**	**13**	**13**	**13**	**13**
parcprinter (50)	39	32	34	32	37	40	41	37	**50**	**50**	**50**	**50**
parking (40)	13	8	8	13	13	2	2	5	13	13	13	**14**
pathways (30)	4	4	4	**5**	**5**	**5**	4	4	**5**	**5**	**5**	**5**
pegsol (50)	**50**	48	48	48	48	48	48	48	**50**	48	**50**	48
petri-net (20)	0	5	4	6	8	16	**18**	**18**	0	8	0	6
pipes-nt (50)	21	23	24	22	23	15	22	18	**25**	23	**25**	**25**
pipes-t (50)	**18**	16	17	17	**18**	13	17	17	**18**	**18**	**18**	**18**
psr-small (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**
rovers (40)	8	8	12	9	10	**14**	13	13	11	12	13	13
satellite (36)	6	6	7	9	9	10	10	**11**	9	9	9	9
scanalyzer (50)	25	37	26	31	**41**	21	21	23	33	39	39	39
snake (20)	**14**	11	13	12	13	10	12	11	13	13	13	12
sokoban (50)	**50**	**50**	**50**	**50**	**50**	46	48	**50**	**50**	**50**	**50**	**50**
spider (20)	**15**	14	**15**	13	14	10	11	11	**15**	14	**15**	**15**
storage (30)	**16**	**16**	**16**	**16**	**16**	15	15	15	**16**	**16**	**16**	**16**
termes (20)	12	12	14	12	13	13	**15**	**15**	13	13	13	13
tetris (17)	10	11	11	10	11	10	12	**13**	**13**	**13**	**13**	**13**
tidybot (40)	22	28	24	28	30	24	29	29	**32**	30	**32**	**32**
tpp (30)	6	7	12	12	10	9	**14**	8	8	11	12	11
transport (70)	35	34	35	35	37	26	28	28	35	36	**39**	38
trucks (30)	9	9	10	13	12	12	10	10	12	13	**16**	**16**
visitall (40)	28	30	**31**	30	30	16	23	24	30	30	30	30
woodwork (50)	25	46	37	38	49	45	46	46	**50**	**50**	**50**	**50**
zenotravel (20)	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**
Sum (1827)	965	1081	1061	1104	1153	1027	1097	1096	1199	1200	1242	**1247**

Table 5: Number of solved tasks by different planners.

		Complementary1	Complementary2	Planning-PDBs	Scorpion				Coverage
					COMB	SYS-SCP	SYS-SCP+COMB	SYS-SCP+CART	
	Complementary1	–	3	5	11	8	8	8	1027
	Complementary2	**26**	–	6	13	12	11	11	1097
	Planning-PDBs	**27**	**9**	–	12	11	10	10	1096
Scorpion	COMB	**29**	**29**	**27**	–	7	3	3	1199
Scorpion	SYS-SCP	**30**	**27**	**26**	**15**	–	4	5	1200
Scorpion	SYS-SCP+COMB	**31**	**29**	**28**	**13**	**12**	–	**6**	1242
Scorpion	SYS-SCP+CART	**30**	**27**	**27**	**14**	**11**	**6**	–	**1247**

Table 6: Comparison of IPC 2018 planners and Scorpion variants.

(SYS-SCP+COMB) is clearly preferable to using either set alone, raising the total coverage by 42–43 tasks. Removing the HC and SYS-2 PDBs from this variant further increases the number of solved tasks to 1247.

5 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 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

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2–6. AAAI Press.

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

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. In McCalla, G. I., ed., *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI 1996)*, volume 1081 of *Lecture Notes in Computer Science*, 402–416. Springer-Verlag.

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.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, 274–283. AAAI Press.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In Edelkamp, S., and Lomuscio, A., eds., *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, 35–50.

Fan, G.; Müller, M.; and Holte, R. 2017. Additive merge-and-shrink heuristics for diverse action costs. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4287–4293. AAAI Press.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Franco, S.; Torralba, Á.; Lelis, L. H. S.; and Barley, M. 2017. On creating complementary pattern databases. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4302–4309. AAAI Press.

Franco, S.; Lelis, L. H. S.; Barley, M.; Edelkamp, S.; Martines, M.; and Moraru, I. 2018. The Complementary1 planner in the IPC 2018. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 28–31.

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.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence* 170(16–17):1123–1136.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733. AAAI Press.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 174–181. AAAI Press.

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

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1–2):9–22.

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

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

Lelis, L. H. S.; Stern, R.; and Sturtevant, N. R. 2014. Estimating search tree size with duplicate detection. In Edelkamp, S., and Barták, R., eds., *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 114–122. AAAI Press.

Moraru, I.; Edelkamp, S.; Martinez, M.; and Franco, S. 2018. Planning-PDBs planner. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 69–73.

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

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3335–3341. AAAI Press.

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.

Pommerening, F. 2017. *New Perspectives on Cost Partitioning for Optimal Classical Planning*. Ph.D. Dissertation, University of Basel.

Rovner, A.; Sievers, S.; and Helmert, M. 2019. 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)*. AAAI Press.

Seipp, J., and Helmert, M. 2014. Diverse and additive Cartesian abstraction heuristics. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 289–297. 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)*. AAAI Press.

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 Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 259–268. AAAI Press.

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

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.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2018. Symbolic perimeter abstraction heuristics for cost-optimal planning. *Artificial Intelligence* 259:1–31.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.