On Creating Complementary Pattern Databases

Santiago Franco[1*], **Álvaro Torralba**[2], **Levi H. S. Lelis**[3], **Mike Barley**[4]
[1] School of Computing and Engineering, University of Huddersfield, UK
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[3] Departamento de Informática, Universidade Federal de Viçosa, Brazil
[4] Computer Science Department, Auckland University, New Zealand
s.franco@hud.ac.uk, torralba@cs.uni-saarland.de, levi.lelis@ufv.br, barley@cs.auckland.ac.nz

Abstract

A pattern database (PDB) for a planning task is a heuristic function in the form of a lookup table that contains optimal solution costs of a simplified version of the task. In this paper we introduce a method that sequentially creates multiple PDBs which are later combined into a single heuristic function. At a given iteration, our method uses estimates of the A* running time to create a PDB that complements the strengths of the PDBs created in previous iterations. We evaluate our algorithm using explicit and symbolic PDBs. Our results show that the heuristics produced by our approach are able to outperform existing schemes, and that our method is able to create PDBs that complement the strengths of other existing heuristics such as a symbolic perimeter heuristic.

1 Introduction

Pattern databases (PDBs) map the state space of a classical planning task onto a smaller abstract state space by considering only a subset of the task's variables, which is called a pattern [Culberson and Schaeffer, 1998; Edelkamp, 2001]. The optimal distance from every abstract state to an abstract goal state is precomputed and stored in a lookup table. The values in the table are used as a heuristic function to guide search algorithms such as A* [Hart *et al.*, 1968] while solving planning tasks. Since a PDB heuristic is uniquely defined from a pattern, we also use the word pattern to refer to a PDB.

The combination of several PDBs can result in better cost-to-go estimates than the estimates provided by each PDB alone. One might combine multiple PDBs by taking the maximum [Holte *et al.*, 2006; Barley *et al.*, 2014] or the sum [Felner *et al.*, 2004] of the PDBs' estimates. In this paper we consider the canonical heuristic function, which takes the maximum estimate over all additive PDB subsets [Haslum *et al.*, 2007]. The challenge is to generate a pattern collection from which an effective heuristic is derived. Multiple approaches have been suggested to select good pattern collections [Haslum *et al.*, 2007; Edelkamp, 2006; Kissmann and Edelkamp, 2011]. Recent work showed that using a genetic algorithm [Edelkamp, 2006] to generate a large collection of PDBs and greedily selecting a subset of them can be effective in practice [Lelis *et al.*, 2016]. However, while generating a PDB heuristic, Lelis *et al.*'s approach is blind to the fact that other PDBs will be considered in the selection process. Our proposed method, which we call Complementary PDBs Creation (CPC), adjusts its PDB generation process to account for the PDBs already generated as well as for other heuristics optionally provided as input.

CPC sequentially creates a set of pattern collections $\mathcal{P}_{sel}$ for a given planning task ∇. CPC starts with an empty $\mathcal{P}_{sel}$ set and iteratively adds a pattern collection $\mathcal{P}$ to $\mathcal{P}_{sel}$ if it predicts that $\mathcal{P}$ will be *complementary* to $\mathcal{P}_{sel}$. We say that $\mathcal{P}$ complements $\mathcal{P}_{sel}$ if A* using a heuristic built from $\mathcal{P} \cup \mathcal{P}_{sel}$ solves ∇ quicker than when using a heuristic built from $\mathcal{P}_{sel}$. CPC uses estimates of A*'s running time to guide a local search in the space of pattern collections. After $\mathcal{P}_{sel}$ has been constructed, all the corresponding PDBs are combined with the canonical heuristic function [Haslum *et al.*, 2007].

We evaluate our pattern selection scheme in different settings, including explicit and symbolic PDBs. Our results show that the heuristics produced by our approach are able to outperform existing methods, and that CPC is able to create complementary PDBs to other existing heuristics.

2 Background

An *SAS*$^+$ *planning task* [Bäckström and Nebel, 1995] is a 4-tuple $\nabla = \langle \mathcal{V}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$. $\mathcal{V}$ is a set of *state variables*. Each variable $v \in \mathcal{V}$ has a finite domain $\mathcal{D}_v$. $\mathcal{O}$ is a set of operators, where each operator $o \in \mathcal{O}$ is a triple $\langle pre_o, post_o, cost_o \rangle$ specifying the preconditions, postconditions (effects), and non-negative cost of o. pre_o and $post_o$ are assignments of values to subsets of variables, $\mathcal{V}_{pre_o}$ and $\mathcal{V}_{post_o}$, respectively. Operator o is applicable to state s if s and pre_o agree on the assignment of values to variables in $\mathcal{V}_{pre_o}$. The result of applying o to a state s is a new *generated* state s' that agrees with $post_o$ on the assignment of values to variables in $\mathcal{V}_{post_o}$ and with s in the remaining variables. We call the *children* of state s the states generated by applying the effects of each applicable operator of s, denoted as $children(s)$. $\mathcal{G}$ is the goal condition, an assignment of values to a subset of variables, $\mathcal{V}_G$. A state is a goal state if it agrees on the assignment of values to the variables in $\mathcal{V}_G$. $\mathcal{I}$ is the initial state, and the

*This work was carried out while S. Franco was a postdoctoral fellow at Universidade Federal de Viçosa, Brazil.

task is to find an optimal (least-cost) sequence of operators from $\mathcal{I}$ to a goal state.

We use A* with an admissible heuristic to search for optimal solutions for ∇; h is admissible if it never overestimates the cost-to-go of states in the problem's space. A* expands a search tree while guided by the function $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the path from $\mathcal{I}$ to s, and $h(s)$ is an estimated cost-to-go from s to the goal.

A node n in a search tree is a data-structure containing a state s, a path from $\mathcal{I}$ to s, and the path's cost $g(n)$. When we refer to a node as a state we mean the state the node represents. An *A* search tree* is defined as the set of nodes generated by A* while solving ∇. Also, we call a *b-bounded search tree* $S(\mathcal{I}, b) = (N, E)$ the set of all paths from $\mathcal{I}$ to a node n with $f(n) \leq b$. Here, N is the set of nodes and E the set of operators where $(n_1, n_2) \in E$ if $n_2 \in children(n_1)$. $S(\mathcal{I}, b)$ might contain multiple nodes representing the same state, i.e., duplicates. By contrast, the A* search tree does not contain duplicates as the algorithm eliminates them.

Pattern databases (PDBs) project the state space onto a subset of variables, called *pattern* $P \subseteq \mathcal{V}$ [Culberson and Schaeffer, 1998; Edelkamp, 2001]. The optimal distance from every abstract state to an abstract goal state is computed prior to the search and stored in a lookup table. This computation is done by performing a backward search in the abstract state space. A PDB size is defined as the cross product of its variable's domains, $\prod_{v \in P} \mathcal{D}_v$.

Symbolic PDBs use symbolic search [McMillan, 1993; Edelkamp, 2002] to perform the backward search. Symbolic backward search uses efficient data-structures such as Binary Decision Diagrams (BDDs) [Bryant, 1986] to succinctly represent sets of states with the same distance to the goal. Contrary to the lookup table of explicit PDBs, the size of the BDDs representing the heuristic does not directly depend on the size of the abstract state space, allowing us to consider larger patterns. To avoid consuming all resources generating a single PDB, the PDB construction can be interrupted if a time or a memory limit are exceeded [Anderson *et al.*, 2007].

A pattern collection $\mathcal{P}$ is a set of patterns which can be combined in order to obtain stronger heuristic functions. Taking the maximum of the h-values of each individual PDB is always guaranteed to be an admissible heuristic, but stronger estimates can be obtained by computing the sum of the h-values. The values of multiple PDBs are additive if no operator affects two of them, where an operator o affects a PDB with pattern P if it has an effect on any of its variables $\mathcal{V}_{post_o} \cap P \neq \emptyset$. A common method to combine multiple PDBs is the canonical heuristic function, which takes the maximum out of all additive combinations of the collection [Haslum *et al.*, 2007]. This idea can be generalized to a cost-partitioning where the cost of each operator is split among the PDBs [Katz and Domshlak, 2008]. We use a restricted form of cost partitioning, called zero-one cost-partitioning [Haslum *et al.*, 2015; Edelkamp, 2006], in which the cost of each operator is only accounted by one PDB. This method creates the PDBs in a pattern collection, ordered from the PDB with most variables to the PDB with the least, setting the cost of an operator to zero in a PDB if it affects any of the previously generated PDBs.

3 Problem Definition

We are interested in finding a set of pattern collections $\mathcal{P}_{sel}$ that minimizes the running time of A* using the heuristic function obtained from $\mathcal{P}_{sel}$, denoted $h_{\mathcal{P}_{sel}}$. We approximate the running time of A* guided by $h_{\mathcal{P}_{sel}}$ while solving a task ∇, denoted $\mathcal{T}(\mathcal{P}_{sel}, \nabla)$, as introduced by Lelis *et al.* [2016].

$$\mathcal{T}(\mathcal{P}_{sel}, \nabla) = J(\mathcal{P}_{sel}, \nabla) \times (t_{h_{\mathcal{P}_{sel}}} + t_{gen}) .$$

Here, $J(\mathcal{P}_{sel}, \nabla)$ is the number of nodes A* employing $h_{\mathcal{P}_{sel}}$ generates while solving ∇, $t_{h_{\mathcal{P}_{sel}}}$ is $h_{\mathcal{P}_{sel}}$'s average time for computing the heuristic value of a single node, and t_{gen} is the node generation time. Although the exact value of $\mathcal{T}(\mathcal{P}_{sel}, \nabla)$ is only known once A* finishes its search, one is able to compute an approximation, denoted $\hat{\mathcal{T}}(\mathcal{P}_{sel}, \nabla)$. The value of $\hat{\mathcal{T}}(\mathcal{P}_{sel}, \nabla)$ is computed by using approximations of $t_{h_{\mathcal{P}_{sel}}}$ and t_{gen}, which are obtained while computing an estimate for $J(\mathcal{P}_{sel}, \nabla)$, denoted $\hat{J}(\mathcal{P}_{sel}, \nabla)$. $\hat{J}(\mathcal{P}_{sel}, \nabla)$ is obtained by running Stratified Sampling [Chen, 1992]. We write $\hat{J}$ instead of $\hat{J}(\mathcal{P}_{sel}, \nabla)$ whenever $\mathcal{P}_{sel}$ and ∇ are clear from the context.

4 Stratified Sampling Evaluation

Stratified Sampling (SS) estimates numerical properties (e.g., tree size) of search trees by sampling. Lelis *et al.* [2014] showed that SS is unable to detect duplicates in the search tree in its sampling procedure. Instead, we use SS to estimate the size of the search tree $S(\mathcal{I}, b)$, for some value b, and use this estimate as an approximation $\hat{J}$ for the nodes generated by A*. SS uses a stratification of the nodes in the search tree rooted at $\mathcal{I}$ through a *type system* to guide its sampling.

Definition 1 (Type System)**.** *Let $S(\mathcal{I}, b) = (N, E)$ be a b-bounded search tree. $T = \{t_1, \ldots, t_y\}$ is a type system for $S(\mathcal{I}, b)$ if it is a partitioning of N. For every $s \in N$, $T(s)$ denotes the unique $t \in T$ with $s \in t$.*

The type system we use accounts for a heuristic h as follows. Two nodes n_1 and n_2 in $S(\mathcal{I}, b)$ have the same type if $f(n_1) = f(n_2)$ and if n_1 and n_2 occur at the same level of S.

SS samples S and returns a set A of *representative-weight* pairs, with one such pair for every unique type seen during sampling. In the pair $\langle n, w \rangle$ in A for type $t \in T$, n is the unique node of type t that was expanded during search and w is an estimate of the number of nodes of type t in S. Since SS is non-deterministic, every run of the algorithm can generate a different set A. We call each run of SS a probe. We refer the reader to SS's original paper [Chen, 1992] for details.

In our pattern selection algorithm we run multiple SS probes to generate a collection of vectors $C = \{A_1, A_2, \cdots, A_m\}$. A vector A^U is created from C by combining all representative-weight pairs in C. For each unique type t encountered in C we add to A^U a representative pair $\langle n, \bar{w} \rangle$ where n is selected at random from all nodes in C of type t, and $\bar{w}$ is the average w-value of all nodes in C of type t. Each entry in A^U represents SS's prediction for the number of nodes of a given type in the search tree.

We run SS with a time limit of 20 seconds and a space limit of 20,000 entries in the A^U structure. SS performs 1,000

probes with $b = h(\mathcal{I})$, where h is CPC's current heuristic function. If SS completes all 1,000 probes without violating the time and space limits, we increase b by 20% and run another 1,000 probes. The process is repeated until reaching either the time or the space limits. The A^U structure is built from the A vectors collected in all probes.

Since our pattern selection approach needs to test multiple heuristics, we run SS once using a type system T defined by CPC's current heuristic and store A^U in memory. Then, $\hat{J}$ is computed for a newly created heuristic h' by iterating over all representative node-weights $\langle n, \bar{w} \rangle$ in A^U and summing the $\bar{w}$-values for which $h'(n) + g(n) \leq b$, where b is the largest value used for probing with SS while building the A^U structure; this sum is our $\hat{J}$ for h'. Also, as mentioned in Section 3, the approximations for $t_{h_{\mathcal{P}_{sel}}}$ and t_{gen} are obtained by measuring them during SS's probes.

5 Adaptable Pattern Collection Generation

Algorithm 1 is a high-level overview of the search CPC performs in the pattern collection space. CPC receives as input a planning task ∇, a base heuristic h_{base} (which could be the h_0 heuristic, *i.e.,* a heuristic that returns zero to all states in the state space), time and memory limits, t and m, respectively, that specify when to stop running CPC. CPC also receives another time limit, t_{stag}, for deciding when the parameters of CPC's search must be readjusted. S_{min} and S_{max} specify the minimum and maximum sizes of the PDBs constructed. We use zero-one cost partitioning on each pattern collection $\mathcal{P}$ so that its PDBs are additive. Once CPC returns a set of pattern collections $\mathcal{P}_{sel}$, we use the canonical heuristic function [Haslum *et al.*, 2007] to combine all the patterns in $\mathcal{P}_{sel}$ into a heuristic function.

CPC creates pattern collections through calls of the function BINPACKINGUCB (see line 5), which we explain in Section 5.1. Once a pattern collection $\mathcal{P}$ is created, CPC evaluates its quality with SS (see line 8), which estimates the running time of A* using a heuristic composed of the patterns already selected by CPC, $\mathcal{P}_{sel}$, added to the new $\mathcal{P}$. If SS estimates that A* solves ∇ faster with a heuristic created from the set of pattern collections $\mathcal{P}_{sel} \cup \mathcal{P}$ than with a heuristic created from $\mathcal{P}_{sel}$, CPC adds $\mathcal{P}$ to $\mathcal{P}_{sel}$ (see line 9). Whenever CPC adds a pattern collection $\mathcal{P}$ to $\mathcal{P}_{sel}$, it performs a local search by applying a mutation operator to $\mathcal{P}$ (see line 7), trying to create other similar and helpful pattern collections (the mutation operator is explained in Section 5.2). If SS estimates that $\mathcal{P}$ does not help reducing A*'s running time, then CPC creates a new $\mathcal{P}$ through another BINPACKINGUCB function call in its next iteration.

The first time EVALUATE-SS is called, CPC runs SS using h_{base} as its type system to create a vector A^U that is used to produce estimates of the A* running time. Whenever a call to EVALUATE-SS returns true, meaning that $\mathcal{P}$ helps reducing A*'s running time, CPC discards A^U and runs SS again with the heuristic constructed from $\mathcal{P}_{sel} \cup \mathcal{P}$ as its type system to generate a new A^U. The intuition behind re-running SS whenever a complementary pattern collection is found is to allow SS to explore parts of the search tree that were not explored in previous runs. Initially, the heuristic used in SS's sampling tend to be weak, and many of the states in the A^U vector SS produces will not be expanded by A* after the new $\mathcal{P}$ is added to $\mathcal{P}_{sel}$. By running SS whenever a better heuristic is constructed, one allows SS to also prune such nodes and focus its sampling on nodes that the current heuristic is not able to prune.

Algorithm 1 Complementary PDBs Creation

Require: Planning task ∇, base heuristic h_{base}, time and memory limits t and m respectively, stagnation time t_{stag}, minimum/maximum PDB size S_{min}, S_{max}.
Ensure: Selected set of pattern collections $\mathcal{P}_{sel}$
1: $\mathcal{P}_{sel} \leftarrow \emptyset$ // *$\mathcal{P}_{sel}$ is a set of pattern collections*
2: $\mathcal{P} \leftarrow \emptyset$ // *$\mathcal{P}$ is a pattern collection*
3: **while** time t or memory m limits are not exceeded **do**
4: **if** $\mathcal{P} = \emptyset$ **then**
5: $\mathcal{P} \leftarrow$ BINPACKINGUCB$(\nabla, S_{min}, S_{max})$
6: **else**
7: $\mathcal{P} \leftarrow$ MUTATION$(\mathcal{P})$
8: **if** EVALUATE-SS$(\mathcal{P}_{sel} \cup \mathcal{P})$ **then**
9: $\mathcal{P}_{sel} \leftarrow \mathcal{P}_{sel} \cup \mathcal{P}$
10: **else**
11: $\mathcal{P} \leftarrow \emptyset$
12: **if** (time since a $\mathcal{P}$ is added to $\mathcal{P}_{sel}$) $> T_{stag}$ **then**
13: adjust S_{min}, S_{max}
14: **return** $\mathcal{P}_{sel}$

5.1 Bin-Packing Algorithms

In this section we describe the methods we consider for generating candidate pattern collections.

Regular Bin-Packing (RBP)

We adapt the genetic algorithm method introduced by Edelkamp [2006] for selecting a collection of patterns. Edelkamp's method, which we call Regular Bin-Packing (RBP), generates an initial pattern collection $\mathcal{P}$ as follows. RBP iteratively selects a unique and random variable v from $\mathcal{V}$ and adds it to a subset B of variables, called "bin", that is initially empty. Once a PDB constructed from the subset of variables in B exceeds a size limit M, RBP starts adding the randomly selected variables to another bin. This process continues until all variables from $\mathcal{V}$ have been added to a bin. Note that since RBP selects unique variables, the bins represent a collection of disjoint patterns.

Once the pattern collection $\mathcal{P}$ is generated, RBP iterates through each pattern p in $\mathcal{P}$ and removes from p any variable not causally related to other variables in p [Helmert, 2004].

Causal Bin-Packing (CBP)

Our CBP approach differs from RBP only in the way it selects the variables to be added to the bins. Instead of choosing them randomly as is done in RBP, CBP selects only the first variable of each bin randomly and then only adds to a bin B variables which are causally related to the variables already in B. In case there are multiple causally related variables to be added, CBP chooses one at random.

We observed empirically that RBP tends to generate pattern collections that result in PDBs of similar sizes, and that

	RBP	CBP	50/50	UCB1
Explicit	908	864	923	**936**
Symbolic	973	909	1,000	**1,009**

Table 1: Planner's coverage while using different bin packing approaches.

CBP tends to generate pattern collections that result in PDBs of various sizes. This is because RBP removes causally unrelated variables after the variable selection is done. By contrast, CBP greedily selects causally related variables as the patterns are created. As a result, usually the first pattern created by CBP will have more variables than all the other patterns created.

Combination of Bin-Packing Approaches with UCB1

We performed a systematic experiment on the optimal STRIPS benchmark suite distributed with the Fast Downward Planning System [Helmert, 2006], which has 1,667 planning tasks, to test CPC with RBP and CBP. That is achieved by replacing the BINPACKINGUCB call in line 5 by RBP and by CBP. The coverage results for the two approaches are presented in Table 1.[1] Although RBP solves more instances than CBP, an inspection of the per-domain coverage results showed that CBP is able to solve many instances that are not solved by RBP, specially in domains with many variables and complex causal graph interactions such as Pipesworld, Tetris, Tidybot, TPP, and Woodworking.

We introduce two approaches for automatically selecting which bin-packing algorithm should be used to create the next pattern collection in CPC. The first is a simple baseline in which every time CPC needs to create a new pattern collection, it randomly chooses between RBP and CBP with equal chance; this method is presented as 50/50 in the table of results. We model the problem of deciding which bin-packing algorithm to use as a multi-armed bandit problem where each algorithm represents an arm. A bin-packing algorithm receives a reward of +1 if it provides a $\mathcal{P}$ that is able to reduce the $\hat{\mathcal{T}}$-value as estimated by SS; the reward is 0 otherwise. We used the UCB1 formula [Auer, 2002], $\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$, to decide which arm (algorithm) to use next. Here, $\bar{x}_j$ is the average reward received by algorithm j, n is the total number of trials made (*i.e.,* calls to a bin-packing algorithm), and n_j is the number of times algorithm j was called. We artificially initialize $\bar{x}_j$ to 10 for all j to ensure that all algorithms are tested a few times before UCB1 can express a strong commitment to a particular option. This helps to reduce the chances of UCB1's selection being unduly influenced by the stochastic nature of the bin-packing approaches.

The coverage results of this approach are shown under UCB1 in Table 1. The baseline 50/50 already outperforms RBP and CBP in terms of coverage. However, the overall best results are obtained by UCB1. This is because UCB1 is able to learn which algorithm works best for a particular problem instance and it allocates more time to the best-performing approach. We use UCB1 in all other CPC experiments in this paper.

[1] All experiments testing individual elements of our proposed technique only differ on the element being tested from our final CPC. We use h_0 as base heuristic in all our experiments unless stated otherwise.

5.2 Mutation Operator

CPC performs mutations on a given pattern collection $\mathcal{P}$ whenever $\mathcal{P}$ is deemed as promising by SS. That is, if SS estimates that $\mathcal{P}$ will not reduce the A* running time, CPC sets $\mathcal{P}$ to $\emptyset$, and in the next iteration of CPC's while loop another $\mathcal{P}$ is created with our UCB approach. On the other hand, if SS predicts that $\mathcal{P}$ is able to reduce A*'s running time, then CPC adds $\mathcal{P}$ to $\mathcal{P}_{sel}$ and, in the next iteration of its while loop, it applies a mutation operator to $\mathcal{P}$, trying to create another pattern collection that might further reduce A*'s running time.

The mutation on $\mathcal{P}$ works as follows. For each bin B (representing a pattern in $\mathcal{P}$), CPC iterates through each variable v in $\mathcal{V}$, and with some probability (mutation rate), if v is already in B, then v is removed from B, otherwise, v is added to B. We use the mutations in pattern collections $\mathcal{P}$ as a way of focusing our search in parts of the pattern selection space that are deemed as promising according to SS. This is because we only perform mutations in pattern collections that SS predicts to complement the pattern collections we have in $\mathcal{P}_{sel}$, *i.e.,* the pattern collections that reduce the value of $\hat{\mathcal{T}}$.

5.3 Dynamic Parameter Adjustment

Some of the instances benefit from a large number of small PDBs, while others require a small number of large PDBs. Thus, instead of fixing the PDB size throughout CPC's pattern selection search, we adjust the size of the PDBs, M, to be constructed during search.

To be specific, if after t_{stag} seconds we are unable to add a new complementary pattern collection to $\mathcal{P}_{sel}$, we increase the size M of the PDBs we generate. The intuition is that if our search procedure does not find complementary patterns for the current PDB size, M, then we assume that this particular planning problem might benefit from larger PDBs.

Optimizing the value of the PDB size M used in the bin-packing algorithms can be difficult. In some cases, there is not a single optimal M value for the planning task. To increase the diversity of PDB sizes, we keep a range $[S_{min}, S_{max}]$ of values, from which M is drawn at random according to a normal distribution. The initial value of S_{min} is 10,000, and the initial value of S_{max} is the maximum problem size, i.e., the cross-product of all the variable's domain sizes.[2] If the search for complementary patterns stagnates, we progressively try larger PDBs. We increase the likelihood of drawing larger M-values from the normal distribution by dynamically increasing the value of S_{min}.

We also performed experiments to test our scheme for automatically adjusting the value of M. Table 2 shows the coverage results when using fixed PDB sizes instead of CPC's scheme for the same 1,667 optimal STRIPS benchmark instances used in the bin packing experiment. The first row specifies the number of PDB entries allowed. The column

[2] We never create explicit PDBs with more than 90,000,000 entries.

	Adjustable	10^4	10^5	10^6	10^7	10^8
Explicit	**936**	891	923	930	918	316
Symbolic	**1,009**	884	910	919	924	940

Table 2: Planner's coverage with our scheme to automatically adjust the PDB size limit and with different fixed values of PDB size limit.

"Adjustable" shows the number of instances the planner is able to solve while using CPC's scheme. The coverage results suggest that the adaptable scheme tends to produce better heuristics than when fixing the value of M. Interestingly, explicit PDBs perform better than symbolic for the smaller PDB sizes because their evaluation (lookup) is faster than that of symbolic PDBs. However, once PDB sizes get large enough, the generation time of symbolic PDBs is on average much smaller than the generation time of explicit PDBs. The performance of explicit PDBs collapses if the allowed size is too large as the PDB quickly exceeds the available memory.

6 Experimental Results

We evaluate CPC on the STRIPS optimal benchmark suite distributed with the Fast Downward Planning System [Helmert, 2006]. All experiments are run under International Planning Competition (IPC) rules for optimal classical planning: 1,800 seconds and 4 GBs of RAM. We use 2.67 GHz Linux sandybridge Xeon CPUs, and all planners we use are implemented within the Fast Downward planning system. All configurations use a h^2-based preprocessor to remove irrelevant actions [Alcázar and Torralba, 2015].

CPC produces explicit (CPC-E) and symbolic (CPC-S) PDBs with a time limit t of 900 seconds and a memory limit m of 2 GB. We assume that both CPC-E and CPC-S receive the h_0 heuristic as h_{base}. To evaluate the ability of CPC to create a complementary heuristic to an informative base heuristic, we include CPC-S-P, which is a version of CPC-S seeded with a symbolic perimeter as base heuristic ($h_{base} = P$). The perimeter is built with time and memory limits of 250 seconds and 1GB of RAM. Once the perimeter is constructed, the remaining time (out of the 900 seconds allowed) is used to find symbolic PDBs which complement the perimeter heuristic.

We compare the coverage of an A*-based planner guided by the heuristics CPC generates with the same planner guided by heuristics generated by alternative methods. Namely, we compare CPC against iPDB [Haslum *et al.*, 2007], GHS combining multiple gaPDBs (denoted as GHS-GA), and GHS combining iPDB, LM-Cut, and gaPDBs (denoted as GHS-ALL) [Lelis *et al.*, 2016]. We also compare CPC with A* guided by symbolic PDBs. We call P the method that builds a perimeter-based heuristic with symbolic search. We also experiment with the Gamer technique for symbolic PDBs [Kissmann and Edelkamp, 2011]. We use Sievers *et al.*'s [2012] implementation of explicit PDBs. All symbolic PDBs use the search enhancements proposed by Torralba *et al.* [2017]. SymBA refers to the winner of the 2014 IPC's optimal track [Torralba *et al.*, 2014; 2016], which uses symbolic bidirectional search with perimeter abstraction heuristics.

Domain	CPC			iPDB	GHS		SymPDB		SymBA
	E	S	S-P		GA	ALL	P	Gamer	
Airport	29	28	27	31	30	**32**	27	26	27
Barman	10	11	11	4	9	6	11	10	**17**
Blocksworld	26	27	30	28	26	28	30	30	**31**
ChildSnack	0	1	2	0	0	0	0	2	**4**
Depot	8	7	8	**9**	7	**9**	7	7	7
Driverlog	14	**15**	14	13	14	14	12	13	12
Elevators	**44**	43	43	43	**44**	**44**	41	37	**44**
Floortile	31	**34**	**34**	16	18	26	**34**	**34**	**34**
Freecell	21	37	33	20	21	20	25	**42**	22
GED	**20**	19	**20**	15	19	19	**20**	19	**20**
Grid	3	3	3	3	3	3	2	**4**	2
Gripper	8	13	**20**	8	8	8	**20**	13	**20**
Hiking	19	**20**	**20**	15	19	18	14	13	15
Logistics	**29**	28	28	25	28	28	20	26	24
Miconic	70	92	106	55	67	**140**	105	107	113
Movie	29	**30**	**30**	**30**	29	**30**	**30**	**30**	**30**
MPrime	23	**25**	24	23	23	24	21	18	23
Mystery	16	15	16	**18**	16	17	15	**18**	15
NoMystery	**20**	**20**	**20**	16	**20**	**20**	14	16	14
Openstacks	49	53	74	49	49	49	75	71	**90**
ParcPrinter	**44**	42	41	36	33	40	37	40	39
Parking	1	1	4	**13**	1	**13**	5	0	5
Path noneg	4	**5**	**5**	4	4	**5**	4	4	**5**
Pegsol	**48**	**48**	**48**	46	**48**	**48**	**48**	47	**48**
PipesworldNT	25	25	24	21	19	20	17	**26**	15
PipesworldT	17	18	18	18	15	17	14	**19**	15
PSR-small	**50**	**50**	**50**	**50**	49	**50**	**50**	**50**	**50**
Rovers	9	13	13	8	8	8	12	13	**14**
Satellite	7	**10**	**10**	6	7	7	**10**	9	**10**
Scanalyzer	23	22	21	23	21	**31**	21	21	21
Sokoban	**50**	**50**	49	**50**	48	**50**	48	49	48
Storage	15	15	15	**16**	15	**16**	15	**16**	15
Tetris	**13**	**13**	**13**	12	9	11	11	**13**	10
Tidybot	23	30	30	25	23	25	25	**35**	19
TPP	12	**15**	**15**	6	7	7	8	8	8
Transport	**39**	35	33	29	38	38	31	34	32
Trucks	10	11	11	9	9	10	10	**15**	12
VisitAll	**33**	**33**	**33**	28	**33**	**33**	18	28	18
Woodworking	31	39	46	25	32	39	46	**48**	**48**
Zenotravel	**13**	**13**	**13**	11	**13**	**13**	9	11	10
Total	936	1,009	**1,055**	857	882	1,016	962	1,022	1,006

Table 3: Coverage of CPC using explicit (CPC-E) and symbolic PDBs (CPC-S), optionally seeded with a perimeter (CPC-S-P). We compare with explicit PDBs (iPDB and GHS), symbolic PDBs (P and Gamer) methods, and SymBA.

6.1 Overall Results

Table 3 shows the coverage results per domain. CPC-S-P solves 1,055 instances and is the best performing system. Comparing the results against the perimeter alone (P), which solves 962 instances, highlights the ability of our method to find complementary PDBs in several domains. The two scatter plots on the left-hand side of Figure 1 further compares CPC-S-P with P. Each point represents one of the 1,667 instances used in our experiment. The x-axis of the first plot represents the number of nodes A* expands while using P, and the y-axis represents the number of nodes A* expands while using CPC-S-P. A* expands substantially fewer nodes when using CPC-S-P. The second scatter plot compares the search time of the A*-based planner using P and CPC-S-P. Again we observe that the planner tends to solve the problem instances more quickly when using CPC-S-P.

Gamer is the second best planner tested, it solves 32 fewer instances than CPC-S-P. CPC-S-P solves more instances than Gamer in 18 domains, the same number of instances in 13 and fewer instances in 9 domains. The symbolic version of CPC that uses $h_{base} = h_0$ (CPC-S) solves 42 fewer instances than Gamer. On the other hand, CPC-S solves more instances than

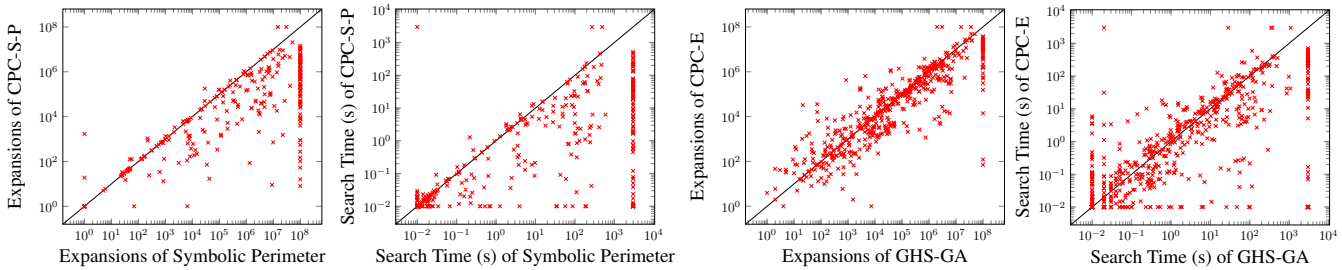

Figure 1: Search time and expanded nodes until last f-layer for CPC-E vs. GHS-GA and CPC-S-P vs. P.

Explicit			Symbolic		
AVG	RS	SS	AVG	RS	SS
872	891	**936**	895	**1,009**	**1,009**

Table 4: Coverage of alternative sampling methods.

CPC-S	P	max(P, CPC-S)	CPC-S-P
1,009	962	1,014	**1,055**

Table 5: Coverage of CPC complementing a perimeter heuristic vs. maximum of regular CPC and the same heuristic.

Gamer in 19 domains, fewer instances in 14 domains and the same number of instances in 7 domains.

Among the methods that only use explicit PDBs, CPC-E outperforms both iPDB and GHS-GA. The comparison with GHS-GA is specially relevant because both methods use a genetic algorithm approach to construct PDBs and SS as evaluation function. The main difference between CPC and GHS-GA is that CPC searches for pattern collections while accounting for the pattern collections that were already selected. This allows CPC to focus its search on parts of the pattern selection space that are deemed as promising by SS for finding complementary pattern collections. By contrast, GHS generates a large number of pattern collections and then tries to select complementary patterns.

The two scatter plots on the right-hand side of Figure 1 compare the number of nodes A* expands as well as the A* search time when using heuristics created by CPC versus those created by GHS with GHS-GA. Although CPC does not substantially reduce the number of node expansions performed by A*, it still reduces the overall search time. This is because the heuristic produced by CPC tends to use fewer pattern collections than the heuristic produced by GHS-GA. Since CPC's heuristic uses fewer pattern collections, its computation will be faster than the heuristic created by GHS-GA, which results in a reduction in the overall search time. The reduction in the overall search time results in more instances being solved. While CPC-E solves 936 instances, GHS-GA solves only 882. GHS-ALL solves more instances than both CPC-E and GHS-GA, and that is mainly due to the use of LM-Cut and its great performance in Miconic.

6.2 Alternative Sampling Methods

Table 4 compares alternative approaches as evaluation functions for guiding the CPC search for pattern collections. Namely, we compare the use of SS with random sampling (RS) and average h-value sampling (AVG). These methods are employed by other PDB generation algorithms: gaPDB [Edelkamp, 2006] uses AVG and iPDB [Haslum *et al.*, 2007] uses RS.

The results in Table 4 suggest that SS is either always better or at least as good as the best sampling method guiding CPC's search through the pattern selection space. When doing explicit PDBs the heuristic CPC produces while guided by SS, A* is able to solve more instances than when guided by the two alternative approaches. For symbolic PDBs, our planner solves the same number of tasks if using either RS or SS. This is because the symbolic PDBs CPC creates tend to be orders of magnitude larger than its explicit PDBs. As a result, the number of symbolic PDBs needed to reduce the size of the search tree A* generates is substantially smaller than the number of explicit PDBs needed to yield similar tree size reduction. Hence, for the symbolic approach, it is more likely that the PDB combination that prunes more nodes during search is also the PDB combination with the fastest lookup time. That is why for symbolic PDBs one is able to obtain similar results while minimizing A*'s search tree size (as is done by RS) and while minimizing the A*'s running time (as is done by SS). Finally, we noticed in this experiment that in some instances A* runs out of memory while using the heuristic created by CPC while guided by SS, and it is able to solve the instances with the alternative approaches. This happens because SS tries to minimize the overall running time of the A* search. By contrast, the alternative approaches are trying to minimize the number of nodes A* expands, which directly correlates to A*'s memory usage.

6.3 Complementarity of CPC

Table 5 shows how our heuristic complements the base heuristic h_{base}. As base heuristic we use the same symbolic perimeter heuristic we use to seed CPC-S-P in Table 3. The first column shows coverage of the symbolic PDB heuristic CPC generates, CPC-S. The second column shows the coverage of the planner using the symbolic perimeter, P, on its own. The third column shows the coverage results of the planner while using a heuristic that takes the maximum of P and CPC-S. The last column shows the coverage results of the planner while guided by a symbolic heuristic created by CPC when $h_{base} = P$.

Table 5 suggests that CPC is able to create a heuristic

Algorithm	iPDB	gaPDB	Gamer	CPC
Evaluation	Random walk	Avg-h	Avg-h	SS
Candidate	HC/HC-VNS	GA	HC	GA'
PDB Size Limit	Fixed	Fixed	—	Dynamic
Aggregation	h^C	$0/1P$	—	$0/1P + h^C$

Table 6: Comparison of PC generation algorithms.

that complements the strengths of P. If we use CPC to create a heuristic CPC-S ignoring that it will be later combined with P (using the maximum), one can solve 1,014 instances. However, if we provide P as CPC's h_{base}, then the resulting heuristic, CPC-S-P, allows one to solve 1,055 instances.

7 Related Work

There are a number of approaches to generate pattern collections $\mathcal{P}$. iPDB [Haslum *et al.*, 2007] performs a hill climbing search (HC) on the space of possible $\mathcal{P}$. It starts with a $\mathcal{P}$ containing a pattern for each goal variable and, at each step, adds a new pattern formed by adding one more variable v to an existing pattern $p \in \mathcal{P}$. To decide which new pattern to generate, it evaluates all valid combinations and selects the one whose PDB has higher h-value in a predetermined sample set of states. iPDB avoids the evaluation of redundant combinations $p \cup \{v\}$ that cannot possibly improve the heuristic value of pattern p according to the causal-graph of the planning task [Haslum *et al.*, 2007; Pommerening *et al.*, 2013]. Finally, Scherrer *et al.* [2015] proposed an extension of iPDB that performs a variable neighborhood search (VNS) [Mladenovic and Hansen, 1997] to escape local minima in the hill climbing search.

GA-PDB [Edelkamp, 2006] starts with initial collections generated by a bin-packing algorithm that distributes the variables among different patterns in each collection. Then, it runs a genetic algorithm (GA) by randomly modifying the existing collection in order to improve it. During search, GA-PDB prefers pattern collections $\mathcal{P}$ with the highest average h-value.

The Gamer planner adapted iPDB's method for symbolic PDBs [Kissmann and Edelkamp, 2011], where there is no limit on the PDB size. Contrary to the other two methods, Gamer generates a single PDB, instead of a collection. To do so, it runs a HC search starting with the PDB that contains all goal variables. At each step it considers adding a new variable, choosing the one that increases the average h-value the most. If several variables increase the average h-value by the same amount, all of them are added to the pattern.

Table 6 summarizes the main characteristics of the previous methods and the alternatives proposed in this paper. We differ on our evaluation function based on SS to select complementary PDBs, on adapting the candidate generation method of gaPDB to increase the quality of the candidates, and on auto-adapting the parameters to dynamically choose the PDB size.

There are multiple ways to combine heuristic estimates that trade off accuracy and computational cost. In this work, we have focused on the PDB generation side, sticking to a relatively simple way of combining the heuristics, *i.e.,* the canonical heuristic function to combine PDBs constructed with a greedy 0-1 operator cost partitioning. There are stronger alternatives that achieve more informative estimates based on more general notions of cost-partitioning [Katz and Domshlak, 2010; Pommerening *et al.*, 2015]. Optimal cost-partitioning for a given state s can be computed in time polynomial in the size of the abstract state spaces [Katz and Domshlak, 2010], though it is often computationally too expensive to pay off in practice. Other practical alternatives that could possibly improve our method exist, such as the post-hoc optimization heuristic [Pommerening *et al.*, 2013] and saturated cost-partitioning [Seipp and Helmert, 2014; Keller *et al.*, 2016; Seipp *et al.*, 2017b]. For more information about cost-partitioning methods, we refer the reader to the recent in-depth analysis by Seipp *et al.* [2017a].

8 Concluding Remarks

In this paper we presented CPC, a new PDB-based heuristic that selects pattern collections with PDBs that are complementary to each other and to other heuristics provided as input. As previous methods, we perform an optimization search on the space of possible pattern collections. Our evaluation function is based on Stratified Sampling, which takes into account previously selected PDBs to guide the search towards complementary patterns. Our experiments show that this guidance often leads to better pattern collections than alternative methods in the literature, such as a method that performs random walk sampling and a method that maximizes the PDBs' average heuristic value.

We also explored different ways to generate candidate patterns during this search. Our best configuration combines two methods, using UCB1 to dynamically learn which method is best for the current task. Similarly, instead of choosing a fixed PDB size limit, we automatically adjust this parameter during the CPC search.

We tested CPC with both explicit and symbolic PDBs. Our results show that CPC compares well against other pattern selection methods. Overall, our best configuration uses CPC to complement a symbolic perimeter heuristic. With this configuration, CPC outperformed all alternative methods tested.

Acknowledgments

Á. Torralba was supported by the German Federal Ministry of Education and Research (BMBF), CISPA grant no. 16KIS0656. S. Franco and L. Lelis were supported by Brazil's CAPES (Science Without Borders) and FAPEMIG. M. Barley was supported by the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AOARD) under award number FA2386-15-1-4069. Thanks to Dr. Pat Riddle for her editorial support.

References

[Alcázar and Torralba, 2015] Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *Proc. ICAPS*, 2015.

[Anderson *et al.*, 2007] Kenneth Anderson, Robert Holte, and Jonathan Schaeffer. Partial pattern databases. In *Proc. SARA*, pages 20–34, 2007.

[Auer, 2002] P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2002.

[Bäckström and Nebel, 1995] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995.

[Barley *et al.*, 2014] Michael W. Barley, Santiago Franco, and Patricia J. Riddle. Overcoming the utility problem in heuristic generation: Why time matters. In *Proc. ICAPS*, 2014.

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

[Chen, 1992] P.-C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[Culberson and Schaeffer, 1998] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Edelkamp, 2001] Stefan Edelkamp. Planning with pattern databases. In *Proc. ECP*, pages 13–24, 2001.

[Edelkamp, 2002] Stefan Edelkamp. Symbolic pattern databases in heuristic search planning. In *Proc. AIPS*, pages 274–283, 2002.

[Edelkamp, 2006] Stefan Edelkamp. Automated creation of pattern database search heuristics. In *Proc. MOCHART*, pages 35–50, 2006.

[Felner *et al.*, 2004] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Haslum *et al.*, 2007] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI*, pages 1007–1012, 2007.

[Haslum *et al.*, 2015] Patrik Haslum, Blai Bonet, and Héctor Geffner. New admissible heuristics for domain-independent planning. In *Proc. AAAI*, pages 1163–1168, 2015.

[Helmert, 2004] Malte Helmert. A planning heuristic based on causal graph analysis. In *Proc. ICAPS*, pages 161–170, 2004.

[Helmert, 2006] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

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

[Katz and Domshlak, 2008] Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *Proc. ICAPS*, pages 174–181, 2008.

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

[Keller *et al.*, 2016] Thomas Keller, Florian Pommerening, Jendrik Seipp, Florian Geißer, and Robert Mattmüller. State-dependent cost partitionings for cartesian abstractions in classical planning. In *Proc. IJCAI*, pages 3161–3169, 2016.

[Kissmann and Edelkamp, 2011] Peter Kissmann and Stefan Edelkamp. Improving cost-optimal domain-independent symbolic planning. In *Proc. AAAI*, pages 992–997, 2011.

[Lelis *et al.*, 2014] Levi H. S. Lelis, Roni Stern, and Nathan R. Sturtevant. Estimating search tree size with duplicate detection. In *Proc. SOCS*, pages 114–122, 2014.

[Lelis *et al.*, 2016] Levi H. S. Lelis, Santiago Franco, Marvin Abisrror, Mike Barley, Sandra Zilles, and Robert C. Holte. Heuristic subset selection in classical planning. In *Proc. IJCAI*, pages 3185–3191, 2016.

[McMillan, 1993] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mladenovic and Hansen, 1997] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & OR*, 24(11):1097–1100, 1997.

[Pommerening *et al.*, 2013] Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the most out of pattern databases for classical planning. In *Proc. IJCAI*, 2013.

[Pommerening *et al.*, 2015] Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In *Proc. AAAI*, pages 3335–3341, 2015.

[Scherrer *et al.*, 2015] Sascha Scherrer, Florian Pommerening, and Martin Wehrle. Improved pattern selection for PDB heuristics in classical planning (extended abstract). In *Proc. SOCS*, pages 216–217, 2015.

[Seipp and Helmert, 2014] Jendrik Seipp and Malte Helmert. Diverse and additive cartesian abstraction heuristics. In *Proc. ICAPS*, pages 289–297, 2014.

[Seipp *et al.*, 2017a] Jendrik Seipp, Thomas Keller, and Malte Helmert. A comparison of cost partitioning algorithms for optimal classical planning. In *Proc. ICAPS*, 2017.

[Seipp *et al.*, 2017b] Jendrik Seipp, Thomas Keller, and Malte Helmert. Narrowing the gap between saturated and optimal cost partitioning for classical planning. In *Proc. AAAI*, pages 3651–3657, 2017.

[Sievers *et al.*, 2012] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. In *Proc. SOCS*, 2012.

[Torralba *et al.*, 2014] Álvaro Torralba, Vidal Alcázar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp. SymBA*: A symbolic bidirectional A* planner. In *IPC 2014 planner abstracts*, pages 105–109, 2014.

[Torralba *et al.*, 2016] Álvaro Torralba, Carlos Linares López, and Daniel Borrajo. Abstraction heuristics for symbolic bidirectional search. In *Proc. IJCAI*, 2016.

[Torralba *et al.*, 2017] Álvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence*, 242:52–79, 2017.