The Complementary2 Planner in the IPC 2018

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

Abstract

This planner is an implementation of the heuristic (CPC) presented in (Franco *et al.* 2017), the only updates are a few bug fixes. This paper contains a brief summary of that work with a slant on which exact configuration was used and why. **Please quote (Franco *et al.* 2017) as well when discussing this planner**.

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 planner we use 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 used symbolic PDBs because the current implementation supports conditional effects, a requirement in the IPC18. Additionally, in our benchmark tests, this was the best option even without conditional effects.

Introduction

This paper contains excerpts from (Franco *et al.* 2017) because this planner is an exact implementation of the CPC heuristic, specifically the CPC-S-P configuration. Other parts of the original paper have been summarized. Comments have been added to reflect the reasoning behind some of our choices. But first, we will give some context information for those not familiar with PDBs.

Excerpt from Original Paper

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 then to find a collection of patterns 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).

IPC2018 Choices

We evaluated our pattern selection scheme in different settings in (Franco *et al.* 2017) , including explicit and symbolic PDBs. Our results showed that combining symbolic PDB heuristics were able to outperform existing methods. Furthermore, it also showed that CPC could create complementary PDBs to other methods. Our best combination was using our method to complement a symbolic perimeter PDB.

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

The selected method to be complemented for this competition first generates a symbolic PDB up to a time limit of 250 seconds, a memory limit of 4GBs[1]. One advantage of starting our algorithm with such a perimeter search is that if there is an easy solution to be found in what is basically a brute force backwards search, we are finished before we even start finding complementary PDBs. If a PDB contains all available variables, any optimal solution for such abstraction is also necessarily an optimal solution in the real search space. In such cases we stop building the perimeter and simply return the optimal plan found.

If no solution is found after the perimeter PDB is finished, our method will start generating pattern collections stochastically until either the generation time limit (900 secs) or the overall PDB memory limit (4 GBs) is reached. CPC decides whether to add a pattern collection to the list of selected patterns if it is estimated that adding such PDB will speed up search. We used the stratified selection time prediction method described in the original paper to estimate this. Note that when a pattern collection is added, all its patterns are collected using the canonical combination method in Fast Downward (from now on referred to as FD as it was in the 2017 version we forked our code from).

Once all patterns have been selected, the corresponding canonical PDB combination is used as an admissible heuristic to do A* search for the sequential optimal track. We also added a cost-bounded option, where we used a slightly modified version of lazy greedy search as coded in FD. The modification is that instead of pruning all generated successor nodes whose g (current path cost) value is above the bounded cost, we actually prune all nodes whose $g + h$ (current path cost + estimated distance to goal) values are above the bounded cost. This is only guaranteed to keep solution cost at or below the bounded cost if the heuristic is admissible. Since this is the case for our heuristic, we take advantage of the improved pruning capability. Note that this track is an experimental version for us, I personally have very little experience in cost-bounded search and make no claim this is the most efficient search method. We thought it would be nice to try the CPC heuristic in this setting as well.

We decided not to submit this planner for the Satisficing track due to the inherent incompatibility of our heuristic with respect to this track. Generating large symbolic PDBs cost a significant amount of time. Finding which patterns make good pattern collections is even more costly because most of the PDBs generated are never used for the actual search. In Satisficing, the critical factor is finding a solution as quickly as possible, and hence it is generally better when using heuristics to pick those which do not incur in large preprocessing costs.

Problem Definition

This section is identical to the original, included for completeness.

[1]A maximum amount of BDD nodes in the perimeter frontier of 10 million was also used. This was used as a failsafe on the actual implementation, otherwise the code occasionally would get stuck while generating the next step for the BDD generation.

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.

Stratified Sampling Evaluation

We used stratified sampling for our planner as described in the original paper. We briefly summarize it here, for a detailed discussion please see (Franco *et al.* 2017).

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.

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'.

Adaptable Pattern Collection Generation

This section is a summary of the original papers, included for completeness.

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 . 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). 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 rerunning 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: P_sel ← ∅ // P_sel is a set of pattern collections
2: P ← ∅ // P is a pattern collection
3: while time t or memory m limits are not exceeded do
4:    if P = ∅ then
5:       P ← BINPACKINGUCB(∇, S_min, S_max)
6:    else
7:       P ← MUTATION(P)
8:    if EVALUATE-SS(P_sel ∪ P) then
9:       P_sel ← P_sel ∪ P
10:   else
11:      P ← ∅
12:   if (time since a P is added to P_sel) > T_stag then
13:      adjust S_min, S_max
14: return P_sel
```

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 in (Franco *et al.* 2017) that RBP tends to generate pattern collections that result in PDBs of

similar sizes, and that 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 UCB1 is a version of UCB whose regret grows logarithmically as a function of the number of actions take. We used this algorithm to choose *in situ* how frequently to use either of both pattern generation algorithms.

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. 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.

In (Franco *et al.* 2017) we performed a systematic experiment on the optimal STRIPS benchmark suite distributed with the FD (Helmert 2006). The coverage results for the two approaches showed using UCB1 to combine both approaches was significantly better than using either one or simply choosing them with equal probability. See the original paper for a more detailed discussion.

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. More details in the original paper.

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.

In the original paper, it was shown that a dynamic range of PDB sizes worked better for our benchmark tests compared to using any of multiple *a priori* fixed sizes.

Empirically-based Choices

1. We used CPC-S-P configuration from the original paper, because it had the overall best results.
2. We only used symbolic PDBs. (Franco *et al.* 2017) because explicit PDBs did not support conditional effects, while symbolic PDBs (as implemented) do. (Franco *et al.* 2017) did not include any domains with conditional effects. Secondly, symbolic PDBs performed significantly better overall for the paper's experiments.[2]
3. We switched to a 64 bits build. After adjusting the size and the maximum number of nodes on the frontier for symbolic PDBs, it was found that more problems were solved, when using the IPC 2018 limits. Both limits were doubled. Limiting the maximum number of nodes in the BDDs frontiers is an implementation failsafe to ensure the memory and time limits are respected.

Results

Following is an ablation-type study were we analyze which components worked best (Table 1). We used the New Zealand Nesi Cluster. Domain names have been abbreviated to either the first 3 letters or the first letter of each word for spaces saving purposes.

Table 1 shows that the combination of bin packed methods (CBP, RBP), aided by an initial perimeter PDB, and regulated by UCB1 (*Comp2/Reg*) was better than any of the individual methods on their own. This confirms our expectations and has a similar behaviour as in (Franco *et al.* 2017) which used all previous IPC domains (seq-opt). Dropping the initial perimeter PDB reduced the overall number of solved problems by 10. Interestingly there was only one domain were the perimeter PDB really helped albeit quite significantly, we would have solved approximately 11 less problems for the Petri Net Alignment (PNA) without the perimeter PDB. Otherwise, the impact of the Perimeter PDB is minimal. In general, the Perimeter PDB works well on domains where getting a good heuristic value is difficult or not that important, e.g. Openstacks. Additionally, if no perimeter is used and we only generate patterns using the RBP generator, 6 fewer problems were solved, the biggest effect is in Snake (Sna) where we solved 3 fewer problems. On the other hand, when only using CBP as a pattern generator, results in no problems lost for Snake but solving 4 fewer Agricola (Agr) and Termes (Ter) problems. However, from previous experiments do note that whether CBP, RBP or a combination of both is the best option is very much dependent on the domain. There is no obvious method to predict *a priori* which bin packing method is best for the current problem.

[2]Note that explicit PDBS can outperform symbolic PDBs on some domains. The reason is that while symbolic PDBs can deal with larger abstractions, they are also more expensive to evaluate. For those domains where it is easy to find a large amount of good quality complementary patterns, explicit can be the better option.

Table 1: Coverage of Complenmentary2 Modules. Reg stands for all components active. NoPer stands for perimeter PDB inactivated. RBP and CBP also have Perimeter inactive.

Domain	Agr	Cal	DN	Nur	OSS	PNA	Set	Sna	Spi	Ter	Total
Comp2/Reg	**6**	**12**	13	**12**	12	**19**	**9**	**14**	11	**16**	**124**
Comp2/NoPer	**6**	**12**	**14**	**12**	12	8	**9**	**14**	11	**16**	114
Comp2/RBP	5	**12**	13	**12**	12	7	**9**	11	**13**	14	108
Comp2/CBP	2	**12**	13	**12**	12	8	**9**	**14**	12	12	106
Competition result bellow included for Completeness											
Comp2	**6**	**12**	12	**12**	**13**	18	9	**14**	12	**16**	**124**

Concluding Remarks

The competition results were quite good for optimal planning, where we were the runners up (winner was 126 problems while we solved 124). We were also the best non-portfolio approach.

Future research avenues, as mentioned in (Franco *et al.* 2017), are using more (or improved) pattern generator methods, keep working on improving *in situ* selection of pattern generators, and analyzing the impact of using online PDBs for selection purposes.

Acknowledgments

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. Thanks to the Fast Downward (and lab testing tool) developers for sharing their code. It is easy to take it for granted, since both have been available for so long, but the constant maintenance and support work is very much appreciated. The complementary1 Planner was build on top of an early 2017 FD fork.

References

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

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

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

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

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

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

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

Santiago Franco, Álvaro Torralba, Levi H. S. Lelis, and Mike Barley. On creating complementary pattern databases. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 4302–4309, 2017.

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.

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.

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

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

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

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

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.