The Complementary1 Planner in the IPC 2018

Santiago Franco[1], Levi H. S. Lelis[2], Mike Barley[3], Stefan Edelkamp[4], Moises Martinez[4], Ionut Moraru[4]
[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
[4] Computer Science Department, Kings College London, United Kingdom
s.franco@hud.ac.uk, levi.lelis@ufv.br, barley@cs.auckland.ac.nz,
stefan.edelkamp@kcl.ac.uk, moises.martinez@kcl.ac.uk, ionut.moraru@kcl.ac.uk

Abstract

This planner is an updated and significantly modified version of the heuristic (CPC) presented in (Franco *et al.* 2017).

The Complementary1 planner uses pattern databases (PDBs). A PDB 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 dynamically creates multiple PDBs which are later combined into a single heuristic function. At a given iteration, our method uses estimates of the search space size to create a PDB that complements the strengths of the PDBs created in previous iterations.

The biggest difference with (Franco *et al.* 2017) is that the original method always started with smaller PDBs and used *a priori* time limits to sequentially increase the PDB's size limit while the new method has no such schedule or initial bias. Complementary1 uses the UCB1 bandit algorithm to learn which PDB size bracket fits best the current problem given the previously selected PDBs. We have also added two new seeding algorithms, based on bin packing, and also added a new pattern generation algorithm based on Gamer. Finally, the code itself has been refactorized to ease the addition of evaluation methods, generation methods and other alternative configurations.

Introduction

Excerpt from (Franco *et al.* 2017)

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 ∇. Regular 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 $\mathcal{P} \cup \mathcal{P}_{sel}$ explores a smaller search space than when using a heuristic built from $\mathcal{P}_{sel}$. CPC uses estimates of A*'s search space to guide a local search in the space of pattern collections[1]. After $\mathcal{P}_{sel}$ has been constructed, all the corresponding PDBs are combined with the canonical heuristic function (Haslum *et al.* 2007).

Comments

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. Futhermore, it also showed that CPC could create complementary PDBs to other methods. Our best combination was

[1]The (Franco *et al.* 2017) CPC version explores both size and time as metrics to decide whether to select a pattern collection. For the time predictions we used SS planner we have re-factorized the code, but did not finish the time selection in time. We have instead used the simpler to code random walk method to evaluate patterns. Note that for symbolic PDBs, both time and size selection methods had very similar performance.

using our method to complement a symbolic perimeter PDB. 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[2]. One advantage of seeding 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.

For this planner, we have added two new seeding methods besides the perimeter PDB we collectively refer to as *bin-packing*. The first one uses First Fit Increasing to try to find the smallest collection of PDB using the bin packing principle. The second method uses First Fit Decreasing to do the same. Bin packing for PDBs tries to create the smallest number of PDBs which uses all available variables. While reducing the number of PDBs used to group all possible variables does not guarantee a better PDB, the less number of collections, the less likely on average to miss interactions between variables due to being placed on different PDBs[3]. PDB selection methods tend to suffer from diminishing returns, i.e.the more time invested using a pattern generation method, the less likely it is to find a new improving one. Using different PDB generation methods or varying their parameters, e.g. PDB size limits, is how we try to ameliorate diminishing returns.

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 (8 GBs) are 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 (Franco *et al.* 2017) 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).

(Franco *et al.* 2017) compared the pattern selection methods to the Gamer algorithm (Kissmann and Edelkamp 2011). Gamer is based on the idea of trying to discover the single best possible PDB for a problem. Its pattern selection method can be summarized as an iterative process, starting with all the goal variables in one pattern, where the casually connected variables who would increase the most the average h value of the associated PDB are added to the Gamer pattern. We have created a new "Gamer-style" pattern generation method which behaves similarly, more details on the next section. This pattern selection method is intended to be complementary to the ones in the original CPC methods.

Once all patterns have been selected, the resulting 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 value is above the bounded cost, we actually prune all nodes whose g+h 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, there is no reason to take advantage of this. Note that this track is an experimental version for us.

We decided not to submit this planner for the Satisficing track due to the inherent incompatibility of our heuristic and the track. Generating large symbolic PDBs cost a significant amount of time. Finding which patterns make good pattern collections is even more costly. In satisficing, the critical factor is finding a solution as quickly as possible, and hence it is generally better when using heuristics to use those which do not incur in large preprocessing costs.

List of Changes and Configuration Choices

- Moved to 64 bits build, due to the increase of memory limit on the IPC to 8 GBs. It required doubling the relevant PDB and overall memory limits.
- Using only symbolic PDBs.
- After the initial Perimeter search is finished, we run two different bin-packing algorithms in order to generate optimized SAS+ variable distribution to generate the PDBs.
  - First Fit Decreasing with a time limit of 50 seconds as recommended by the Authors. This algorithm distributes the variables in different bins according to their size in bits. The variables are initially shorted by their size. Then the smaller variables are grouped in the first bins, while the bigger are grouped in the last ones and sometimes on their own.
  - First Fit Increasing with a time limit of 75 seconds. This algorithm distributes the variables in different bins according to their size as the previous one but in this case the bigger variables are grouped in the first bins while the smaller are grouped in the last ones. The authors empirical tests have showed First Fit Increasing to do better on average when compared to the Decreasing version of the algorithm.
- Dropped (Franco *et al.* 2017) Stratified Sampling (*SS*), we are still finishing porting the original SS code. Note that for symbolic PDBs performance was quite similar. This is explained in (Franco *et al.* 2017), to summarize on average when adding a symbolic PDB[4] which reduces the size of the search space it also tends to reduce the overall

[2] 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.

[3] The packing algorithm used here ensures that each PDB has a least one goal variable and also that all variables in a PDB are casually connected, on their own or through a chain of local variables, to at least one goal variable in the PDB.

[4] Because on average symbolic PDBs enables us to cover much larger search spaces, hence reducing the benefit of using multiple complementary smaller PDBs vs a few larger ones.

run time, hence making both evaluation methods almost equivalent performance wise.

- All PDB size limits (from a minimum of 10^8 to a maximum of 10^{20}) are equally likely to be chosen. We use the UCB1 algorithm to learn *in situ* which PDB size limits are likeliest to do better. Note that the UCB1 will change the recommended PDB size limits if diminishing returns become a significant problem for a specific PDB size bracket. On the original version, fixed time limits were given to increase the PDB size limit by an order of magnitude, potentially forcing the heuristic to keep trying a size limit not justified by the current problem data.
- New Gamer-inspired generation method. Our modified CPC algorithm decides on each iteration which pattern generation method to use. We use UCB1 to learn whether to use the CBP (Franco *et al.* 2017) generation method or the Gamer-inspired method. Note that the Gamer algorithm has a termination condition if no variable can be added to improve the average heuristic value of the selected pattern. In our case, we stop calling the Gamer generation method if we have also discovered that no variable can sufficiently increase the average heuristic value given the current time and size limits.
- Hybrid evaluation methods: Our other pattern generation methods start from scratch, however for the Gamer style pattern selection method, the choice is always whether to add variables to the previously selected pattern. For the Gamer-inspired pattern generation method, we use the average heuristic values to decide whether the next iteration is improving the pattern. If no variable can be added which sufficiently increases the average heuristic value of the Gamer style pattern, this method is dropped from the available pattern selection methods UCB1 can select from. However, in terms of comparing the Gamer style pattern with the already selected patterns by CPC we still use the *in situ* probing mechanism based on problem data, in this case whether the size of the search space is predicted to be reduced by adding the new Gamer style pattern.
- UCB1 is also used to decide how many goal variables are present in a single pattern, the original CBP method was seeded by just one goal variable per pattern. We noticed that one of the reasons Gamer does so well for some problems is that it starts with all the goal variables. For some problems, missing even one goal variable in each pattern when using CBP results in much lower accuracy. We use UCB1 to learn if this is the case in the current problem. As an added bonus, it increases the diversity of PDB generation methods we use and hence hopefully ameliorate diminishing returns.

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.

Complementary1, the newest implementation, solved the same number of overall problems as Complementary2, the same implementation as in (Franco *et al.* 2017) with some bug fixes, however their ablation studies tell a rather different study, A full analysis goes beyond the scope of this extended abstract, instead we are going to provide our first impressions.

Table 1 shows the overall results for our competition submission (Comp1/Reg), with and without the initial perimeter PDB (Comp1/NoPer), with and without the seeding bin packing generator and finally for each of the individual packing method on their own (CBP, Gamer).

The biggest difference with Complementary2 is that dropping the initial perimeter PDB would have increased the number of overall solved problems by 2 [5]. The initial perimeter PDB was very helpful for (Franco *et al.* 2017) and it was also helpful for the same implementation (Complementary2 planner) for the IPC2018 problems. Dropping the perimeter PDBs resulted in solving at least 10 problems less when using any combination of the pattern generators used in Complementary2. It seems that using UCB1 to learn which PDB sizes fit best the problem has reduced our dependency on the perimeter PDB to obtain best results. For example, Complementary2/CBP solved 106 problems while Complementary1/CBPNoBP solved 120. Both used the same pattern generation method(albeit with the option to choose more starting goal variables in the latest version of CBP). Neither used a perimeter PDB. For detailed Complementary2 results, see the Complementary2 extended abstract.

It was also surprising to us that the Gamer module, seeded by BinPack (Comp1/Gamer+BinPack), solved 4 more problems than if we use our selection mechanism (Comp1/NoPer+BinPack) and would have actually won the competition. This would indicate that for some problems the best option was to keep growing the Gamer-style pattern but instead we selected CBP (or run out of time before we could grow the Gamer-style PDB to the same size). Finding out which is future work.

Finally, we also included the best possible results if we knew which is the best pattern generator method *a priori* for free. 139 problems are actually solvable using the right combination of generation methods, 15 more than when using our selection mechanism. Of course this comparison is biased,i.e. when running our selection mechanism the whole available time has to be split between the preferred pattern generators, while when picking the best out of each methods, each of them had the whole 900 seconds generation time. This means the number of patterns tested is much larger when giving each method 900 seconds. What it does show is the potential of symbolic PDBs in planning.

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

[5] and hence would have draw with the competition winner in terms of problems solved

Table 1: Coverage of Complementary1 Modules. Reg stands for all components active. "NoPer" stands for perimeter PDB inactivated. "+BinPack" stands for using PDBs generated by bin packing generator. "CBP", "Cgamer" and "BinPackOnly" rows also have Perimeter inactive.

Domain/Method	Agr	Cal	DN	Nur	OSS	PNA	Set	Sna	Spi	Ter	Total
Comp1/Reg	10	11	13	12	**12**	**19**	**9**	10	**12**	**16**	124
Comp1/NoPer+BinPack	10	**12**	**14**	**14**	**12**	6	**9**	12	11	**16**	126
Comp1/NoBinPack	6	11	13	**14**	**12**	**19**	**9**	11	11	**16**	122
Comp1/CBP+BinPack	8	**12**	**14**	13	**12**	**18**	**9**	11	11	**16**	124
Comp1/CBP-NoBinPack	6	**12**	**14**	13	**12**	18	**9**	9	11	**16**	120
Comp1/Gamer+BinPack	**13**	**12**	**14**	**14**	**12**	17	**9**	12	11	**16**	**130**
Comp1/Gamer-BinPack	8	12	12	**16**	**12**	18	**9**	**14**	11	**16**	128
Comp/BinPackOnly	7	**12**	14	12	**12**	7	**9**	11	11	12	107
Solved by any of the Comp1 methods above											
*	14	12	14	16	12	20	9	14	12	16	139
Competition result below included for Completeness											
Comp1	10	11	14	**13**	13	17	8	11	**11**	**16**	124

non-portfolio approach. However, a Gamer-style approach would have resulted in the best overall results (and easily won the competition). This is not that surprising when looking at (Franco *et al.* 2017) where even though our selector method (without gamer generators) did better overall, it was the second best method. This confirms that which Pattern Generator method is better is very much a question of which domain is it to be used for.

Interestingly, when running each pattern generation method on its own, 15 more problems are solvable. The question is if this was because there was more time allocated to generate patterns when running each method on their own, or if the selection mechanism has significant scope for improvement. This is future research.

Using the 2 bin-packing generators to seed the heuristic proved quite useful. It improved our overall results for all the methods we tested. The implementation we did of the Bin Pack generator was very last minute and did not include any stochastic method. Generally, the more diverse the pattern generators used, the more likely it is for one of them to find good patterns. The preliminary results here seem to indicate that using bin packing techniques as pattern generators is quite promising as a complement to CBP (or RBP in Complementary2) and Gamer.

Finally, it seems that Complementary1 using the UCB1 bandit algorithm to learn which PDB size bracket fits best the current problem, given the previously selected PDBs, has resulted in lowering our dependency on the perimeter PDB to obtain best results. While on the old implementation (Complementary2) dropping the perimeter would result in solving 10 fewer problems, Complementary1 actually solves more problems without the perimeter PDB, regardless of the combination of pattern generators used. A more extensive and detailed analysis is required to confirm this first impression.

Acknowledgments

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

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

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.

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, 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.