Stronger Abstraction Heuristics Through Perimeter Search

Patrick Eyerich
University of Freiburg, Germany
eyerich@informatik.uni-freiburg.de

Malte Helmert
University of Basel, Switzerland
malte.helmert@unibas.ch

Abstract

Perimeter search is a bidirectional search algorithm consisting of two phases. In the first phase, a limited regression search computes the *perimeter*, a region which must necessarily be passed in every solution. In the second phase, a heuristic forward search finds an optimal plan from the initial state to the perimeter.

The drawback of perimeter search is the need to compute heuristic estimates towards *every* state on the perimeter in the forward phase. We show that this limitation can be effectively overcome when using *pattern database* (PDB) heuristics in the forward phase.

The combination of perimeter search and PDB heuristics has been considered previously by Felner and Ofek for solving combinatorial puzzles. They claimed that, based on theoretical considerations and experimental evidence, the use of perimeter search in this context offers "limited or no benefits". Our theoretical and experimental results show that this assessment should be revisited.

Introduction

Perimeter search (Dillenburg and Nelson 1994) is a classical bidirectional state-space search algorithm. It is parameterized with a natural number $r > 0$ we call the *radius* and proceeds in two phases.

The first phase, the *backward phase*, generates all states from which the distance to the nearest goal state is r. This set of states is called the *perimeter* (for radius r). It is traditionally generated through a backward breadth-first search from the goal state(s) with complete duplicate elimination. As a special case, if the initial state of the problem is reached during the backward phase, the optimal solution to the problem can be reported directly.

The second phase, the *forward phase*, performs a normal forward search using a heuristic search algorithm such as IDA* or A* with two modifications. Firstly, the search terminates as soon as a state on the perimeter is expanded. The optimal solution then consists of the path to the perimeter state that was found, followed by a shortest path from there to the closest goal state (which can either be stored during the backward phase or recomputed if memory is scarce). Secondly, the heuristic estimate for state s measures the distance to the closest perimeter state rather than to the goal: $h_{\text{Per}}(s) = \min_{s' \in \mathit{Per}} h(s, s')$, where *Per* denotes the perimeter. This requires a heuristic function that can estimate distances between arbitrary states, not just distances to the goal.

Bidirectional heuristic search algorithms like perimeter search were intensively studied in the 1990s (e.g., Manzini 1995; Kaindl and Kainz 1997). Since then, research activity in this area has noticeably cooled off. In classical planning, heuristic search algorithms represent the state of the art, but bidirectional algorithms are rarely pursued.

A major reason why perimeter search is not used more frequently is the need to compute distances towards every state on the perimeter during each state evaluation, which appears expensive.[1] In some cases, however, it is possible to circumvent this overhead through precomputations. In particular, with *pattern database (PDB) heuristics* (Culberson and Schaeffer 1998), we can seed a single pattern database with all perimeter states, perform a single search through the abstract state space defined by the pattern, and then only perform a single PDB lookup for each state evaluation.

The combination of perimeter search and PDB heuristics has been considered previously by Felner and Ofek (2007) for solving combinatorial puzzles. Their evaluation of the approach is quite bleak. Based on theoretical observations and experiments, they report that the discussed approach offers "limited or no benefits" over a simpler approach that uses a regular PDB seeded with the goal and only uses information from the backward search to increase heuristic estimates of states known to be outside the perimeter to $r + 1$.

In this paper, we reconsider the combination of PDB heuristics with perimeter search. We do so in the context of classical planning, where PDB heuristics have been previously successful (e.g., Edelkamp 2001; Haslum, Bonet, and Geffner 2005; Haslum et al. 2007; Sievers, Ortlieb, and Helmert 2012). We show that despite several additional difficulties in the planning context that are not present in the combinatorial puzzle setting, perimeter PDBs can significantly reduce the search effort compared to regular forward-search PDB heuristics.

[1] Manzini (1995) describes an optimization that avoids some of these computations, but this does not change the situation fundamentally.

Background

A planning task $T = \langle V, O, s_\mathrm{I}, s_\star \rangle$ consists of a set of finite domain variables V, where each variable $v \in V$ is associated with a domain $\mathcal{D}_v$; a set of operators O; an *initial state* s_I that is given by a variable assignment (a *state*) over V; and a set of goal states $s_\star$ that is defined by a partial variable assignment (a *partial state*) over V. Analogously to the Boolean setting we identify (partial) states with the set of *atoms* that they make true. An atom is of the form $(v\!:\!d)$ where v is a variable and $d \in \mathcal{D}_v$ is a value. We refer to the variable v of an atom $x = (v\!:\!d)$ with *var*(x). We write $s[v]$ for the value associated with variable v in state s.

An *operator* $o = \langle \mathit{pre}, \mathit{eff} \rangle \in O$ consists of preconditions *pre* and effects *eff*, both of which are partial states. We say that o is *applicable* in state s if $\mathit{pre} \subseteq s$. The *application* $\mathit{app}(o, s) = s'$ of o in s yields the state s' such that $s'[v] = d$ if there is an effect $(v\!:\!d) \in \mathit{eff}$ and $s'[v] = s[v]$ if *eff* contains no atom with variable v. A state s' is *reachable* from a state s if there are sequences of operators $\langle o_1, \dots, o_n \rangle$ and states $\langle s_0, \dots, s_n \rangle$ such that $s_0 = s$, o_i is applicable in s_{i-1} and $\mathit{app}(o_i, s_{i-1}) = s_i$ for all $i = 1, \dots, n$. The *cost* of s is the minimal n for which such sequences from s to a state $s_n \supseteq s_\star$ exist. The objective of optimal classical planning is to find such a sequence starting from s_I.

A *pattern* $P \subseteq V$ is a subset of variables. Let $f_P(x)$ be a function that syntactically removes all atoms a with *var*$(a) \notin P$ from x where x is a partial state or operator. Given planning task $T = \langle V, O, s_\mathrm{I}, s_\star \rangle$, the abstract planning task for pattern $P \subseteq V$ is given by $T|_P = \langle P, \{f_P(o) \mid o \in O\}, f_P(s_\mathrm{I}), f_P(s_\star) \rangle$. A *pattern database* (PDB) PDB^P for P is a mapping from the states over P to their cost in $T|_P$. PDBs induce admissible heuristics: $\mathit{PDB}^P(f_P(s))$ never exceeds the cost of s for all states s of T.

Perimeter Search for Planning

When applying perimeter search to planning with pattern databases, there are several challenges that are not present in the combinatorial puzzle setting:

1. Existing work on perimeter search focuses on settings with one goal state and invertible operators. In that case, the backward search to compute the perimeter is structurally identical to forward search from the initial state and does not require any special measures. In planning, there may be trillions of goal states, and a state can have a huge number of predecessors via the same operator.
2. In the heuristic search literature, it is often assumed that PDB construction time can be amortized over many runs of the search algorithm. In planning, typically every planner run faces a different search space, so that PDB computation must be amortized within a single run. The combination with perimeter search aggravates this issue because we must match every abstract state of the PDB to every state on the perimeter to seed the PDB.
3. In planning, it is usually assumed that no instance-specific parameter tuning takes place, so we must find a suitable automated way to set the perimeter radius.

We now discuss how we address these challenges.

Backward Search

Of the three challenges, dealing with huge state sets in the backward search is the most pressing one: without a suitable mechanism to deal with it, perimeter search is inapplicable to many planning tasks. Fortunately, a well-known solution to this issue exists in the form of *regression*. Formally, the regression of a set of states S' over an operator o, *regr*(o, S'), is the set of states in which o is applicable and leads to a state in S'. In other words, $s' = \mathit{app}(o, s)$ iff $s \in \mathit{regr}(o, \{s'\})$.

Let us say that a partial state s *represents* all states $\tilde{s}$ with $s \subseteq \tilde{s}$. An important structural property of our planning formalism is that whenever S' is a state set that can be represented by a partial state s', then *regr*(o, S') can also be represented by a partial state, and this partial state can be efficiently computed from o and s'. Because the set of goal states is also represented by a partial state (namely $s_\star$), we can efficiently perform a search in *regression space*, where each search node is associated with a partial state that represents a set of states backward-reachable from the goal.

Overall Algorithm Algorithm 1 shows how such a search can be implemented. Function *regressionSearch* iteratively conducts a breadth-first search that starts with the goal (line 3) and then explores one layer at a time (line 5). At any time, the search layer of the last completed iteration forms the perimeter that is returned upon termination (line 9).

Using a fixed perimeter radius is not a good option in a general planning system because the number of regressable operators varies too widely between tasks. Since we cannot manually tweak the radius on a per-task basis, the algorithm includes three different termination criteria: a maximal radius, a runtime limit and a memory bound (lines 4, 6, 17).

Efficiency Improvements To amortize perimeter PDB construction within a single planner run, we must incorporate techniques that reduce the runtime as much as possible. Here we describe some of the most important optimizations. Taken together, they frequently reduce the time for computing the perimeter PDB by several orders of magnitude.

Two simple but effective optimizations are to check if the regression of an operator o over partial state s results in an inconsistent state before attempting to compute the regressed partial state (line 16) and to avoid the construction of regressed states that are dominated by their parent (line 19), which is a very common occurrence.[2] We also use mutex information computed by the translator component of Fast Downward (Helmert 2009) to prune partial states which only represent states that are known to be unreachable from the initial state of the forward search (line 22).

The most important and most complex optimization is related to duplicate detection. Detecting duplicate nodes during backward search is vital to keep runtime and memory usage in check. It also tightens the perimeter: without eliminating duplicates, we only know that all states in breadth-first search layer k have a goal distance of *at most* k. It is preferable, however, to seed a perimeter PDB only with

[2]Partial state s dominates partial state s' iff $s \subseteq s'$. In this case, the states represented by s form a superset of the states represented by s', and hence all states represented by s' are duplicates.

states with a goal distance of *exactly* k. While the resulting perimeter PDBs are admissible in either case, tighter perimeters lead to larger heuristic values.

Detecting duplicates is much more challenging in regression search than in progression search because detecting exact matches (as in progression search, which use hash tables for this purpose) does not suffice. Instead, we want to detect whether a newly generated partial state is *dominated* by any previously generated partial state, which involves subset queries over large sets of partial states (line 24).

We address this problem by using a data structure based on the *successor generators* of Fast Downward (Helmert 2006, Section 5.3.1). We refer to this data structure as *match trees* here, since the original name is not descriptive in our context. Roughly speaking, a match tree stores a set of partial states (in Fast Downward: operator preconditions; here: partial states previously encountered in the regression) and allows efficient subset queries: is any of the partial states stored in the tree a subset of a given query state s?

We extended match trees in two ways. Firstly, the trees in Fast Downward must be queried with full states. We extended them to allow partial query states. Secondly, the original trees are built for a fixed set of partial states known a priori. We must build match trees incrementally, adding partial states as they are added to the next perimeter (line 26).

We remark that there are certain cases where we do not detect duplicates in Algorithm 1 since (for efficiency reasons) we do not remove regression states that have already been previously added to the next perimeter. If two partial states s and s' with $s \subseteq s'$ are generated in the same search layer, then the dominance relationship between the two is only detected if s is processed before s'. We expect that the impact of this lost opportunity is low since it only affects partial states in the same search layer – all dominance relationships with respect to previous layers are found.

Algorithm 1: Regression search.

```
1  def regressionSearch()
2      generateRegressionOperators()
3      perimeter ← {goal}
4      for depth = 1 to maxRadius do
5          nextPerimeter ← extendPerimeter(perimeter)
6          if nextPerimeter = failure then
7              break
8          perimeter ← nextPerimeter
9      return perimeter

10 def extendPerimeter(perimeter)
11     nextPerimeter ← ∅
12     for s ∈ perimeter do
13         if s matches originalInitState then
14             reportPlan(s)
15             exit
16         forall operators o that are regressable in s do
17             if time or memory limit reached then
18                 return failure
19             if s dominates regression of o in s then
20                 continue
21             s′ ← regression of o in s
22             if s′ violates a mutex then
23                 continue
24             if s′ dominated by a known partial state then
25                 continue
26             nextPerimeter.add(s′)
27     return nextPerimeter
```

Computing Perimeter PDBs

The PDB for a pattern $P \subseteq V$ in a regular forward search algorithm is constructed by seeding all abstract states (states over P) that match $f_P(s_\star)$ to 0 and performing a backward breadth-first search from these states to all abstract states.

The basic idea of incorporating a perimeter into PDBs is to avoid initializing the abstract goal states to 0. Instead, for all perimeter states s, we seed $f_P(s)$ to the perimeter radius r. This is again followed by a backward breadth-first search as in the regular approach. In the following, we refer to a PDB that is computed in such a way as a *perimeter PDB*.

Properties of Perimeter PDBs To guarantee optimal solutions, perimeter PDBs must be admissible for states *on or outside* the perimeter, i.e., with cost at least r. States inside the perimeter do not matter, as they are never evaluated by the perimeter search algorithm (which stops the regular search as soon as a state on the perimeter is expanded).

For states on or outside the perimeter, the perimeter PDB values are indeed admissible estimates. To see this, consider a modified problem where the goal states are exactly the perimeter states. (Such a goal set is not representable in our planning formalism, but this does not matter.) A *regular* PDB for this problem would seed exactly the same states (to 0) that the perimeter PDB of the original problem seeds to r. Since regular PDBs are admissible, and taking into account that solving the original problem requires finding a shortest path to the perimeter and then taking r more steps towards the goal, this implies that perimeter PDBs are admissible.

It is also easy to see that each entry of a perimeter PDB is at least as large as the corresponding entry of a regular PDB. Consider an abstract state s_P for which the perimeter PDB is seeded to r. This means that there exists a perimeter state s with $f_P(s) = s_P$. Then the regular PDB must have a value of *at most* r for s_P: otherwise it would be inadmissible for s, contradicting the known properties of PDBs. We conclude that perimeter PDBs dominate regular PDBs.

Efficient Seeding Seeding perimeter PDBs can be a major bottleneck of the overall algorithm. Assume that we are constructing a PDB with 10^6 states and a perimeter with 10^5 partial states. A naive implementation would test for each PDB entry if it matches one of these partial states, requiring 10^{11} tests. This can again be improved with match trees, but there is a better solution: rather than iterating over the PDB states, iterate over the partial states on the perimeter and systematically construct all (and only the) PDB states that match this pattern. This can even be done directly on the level of the perfect hash values used to index PDBs, following ideas by Sievers, Ortlieb, and Helmert (2012).

	AIRPORT	BLOCKS	DEPOT	DRIVERLOG	FREECELL	GRID	GRIPPER	LOGISTICS00	LOGISTICS98	MICONIC	MPRIME	MYSTERY	OPENSTACKS	PATHWAYS	PIPES-NT	PIPES-T	PSR-SMALL	ROVERS	SATELLITE	TIDYBOT	TPP	TRUCKS	VISITALL	ZENOTRAVEL	**SUM**
PDB	**21**	19	6	10	**16**	2	7	13	3	51	23	16	7	4	**15**	**16**	49	6	6	**10**	6	6	9	8	329
PPDB-1-∞-∞	**21**	19	6	10	**16**	2	7	13	3	51	23	16	7	4	**15**	**16**	49	6	6	**10**	6	6	9	8	329
PPDB-3-∞-∞	**21**	21	6	10	**16**	2	7	13	3	52	**27**	**17**	7	4	**15**	9	49	6	6	9	6	6	9	**9**	330
PPDB-5-∞-∞	**21**	21	6	10	14	2	7	13	3	54	18	13	7	4	5	6	49	6	6	0	6	6	9	8	295
PPDB-∞-8-1024	18	**28**	6	**11**	15	2	7	13	3	**56**	**27**	**17**	7	4	**15**	13	**50**	6	6	**10**	6	6	9	**9**	344
PPDB-∞-8-256	**21**	27	6	**11**	**16**	2	7	13	3	55	**27**	**17**	7	4	**15**	**16**	**50**	6	6	**10**	6	6	9	**9**	**349**

Table 1: Solved problems per domain. PDB refers to a regular PDB heuristic. PPDB-r-t-m refers to the perimeter PDB approach with a maximum radius of r, time limit of t minutes and memory limit of m MB for the backward phase.

Forward Search

Our final remark concerns the forward phase of perimeter search. As discussed before, it must terminate when a state on the perimeter is expanded, which requires efficiently testing membership in the perimeter. If we consider that the perimeter can comprise 10^5 or more partial states, a naive approach is prohibitively expensive. We address this problem by only performing the check on states whose heuristic value equals r (a necessary criterion for being on the perimeter) and by again using match trees.

Experiments

We implemented the perimeter PDB approach (*PPDB*) in the planning system Fast Downward (Helmert 2006). Table 1 shows coverage on all uniform-cost benchmarks of the optimization tracks of past International Planning Competitions. Our experiments were run on 2.3 GHz AMD Opteron processors with a 2 GB memory limit of and a 30 minute timeout. All configurations use a maximal PDB size of 10^6.

Regular PDBs and PPDBs with a fixed radius of 1 solve the same set of tasks. With a fixed radius of 3, PPDB is able to solve 9 more tasks than PDB across five domains. However, it also loses 8 tasks in two other domains. The reason for this worse performance is that the backward phase does not terminate within the time and memory bounds. This effect becomes more apparent with larger radius: PPDB with a fixed radius of 5 solves 34 fewer problems than PDB.

Replacing the fixed radius by a time and memory limit for the regression search solves this problem (see last two rows of Table 1). The resulting radii of the perimeter vary widely, from as little as 1 to more than 70. In the configuration that allows a larger perimeter (memory limit 1 GB for backward phase), another problem emerges in some domains: even with the efficient implementation mentioned in the previous section, checking the states of the forward search against the perimeter becomes a bottleneck. This effect is very noticeable in the AIRPORT domain, where increasing the memory bound from 256 MB to 1 GB loses 3 tasks.

With empirically determined limits of 8 minutes and 256 MB, PPDB solves 20 more tasks than PDB, a nice improvement considering the exponential growth in difficulty for these benchmark sets. As expected, PPDB works best in domains with a low branching factor during regression search like BLOCKS. Figure 1 compares the number of expansions of PDB and perimeter PDB on all benchmark tasks and complements the theoretical dominance result well.

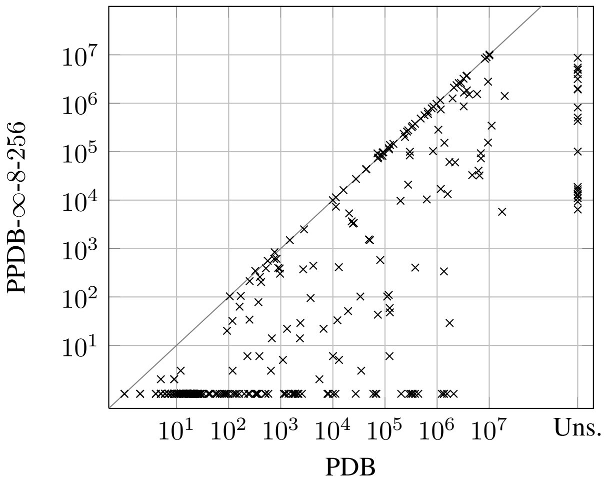

Figure 1: Expansions of the forward phase, comparing PDB to PPDB with 8 min/256 MB backward search limits.

Conclusion

We investigated the combination of pattern databases and perimeter search for optimal classical planning. We argued that PDBs seeded with the states of a perimeter are more accurate than traditional PDBs, which is confirmed by our experiments. We also discussed how to efficiently implement perimeter search for classical planning and experimentally showed that the increased informativeness of perimeter PDBs outweighs the overhead of the approach.

In future work we plan to extend perimeter PDBs to settings of non-uniform action costs, incorporate it to the iPDB approach of Haslum et al. (2007) and revisit its use for classical heuristic search domains.

Acknowledgments

This work was supported by the German Aerospace Center (DLR) as part of the project "Kontiplan" (50 RA 1221) and by the Swiss National Science Foundation (SNSF) as part of the project "Abstraction Heuristics for Planning and Combinatorial Search" (AHPACS).

References

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

Dillenburg, J. F., and Nelson, P. C. 1994. Perimeter search. *Artificial Intelligence* 65:165–178.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, 13–24.

Felner, A., and Ofek, N. 2007. Combining perimeter search and pattern database abstractions. In Miguel, I., and Ruml, W., eds., *Proceedings of the 7th International Symposium on Abstraction, Reformulation and Approximation (SARA 2007)*, volume 4612 of *Lecture Notes in Artificial Intelligence*, 155–168. Springer-Verlag.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, 1163–1168. AAAI Press.

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

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* 7:283–317.

Manzini, G. 1995. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence* 75(2):347–360.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient implementation of pattern database heuristics for classical planning. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SOCS 2012)*, 105–111. AAAI Press.