Efficient Implementation of Pattern Database Heuristics for Classical Planning

Silvan Sievers and **Manuela Ortlieb**
University of Freiburg, Germany
{sievers,ortlieb}@informatik.uni-freiburg.de

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

Abstract

Despite their general success in the heuristic search community, pattern database (PDB) heuristics have, until very recently, not been used by the most successful classical planning systems.

We describe a new efficient implementation of pattern database heuristics within the Fast Downward planner. A planning system using this implementation is competitive with the state of the art in optimal planning, significantly improving over results from the previous best PDB heuristic implementation in planning.

Introduction

Heuristics based on *pattern databases* (PDBs), originally introduced by Culberson and Schaeffer (1996; 1998), are a staple of the heuristic search literature. In *automated planning*, their use has been pioneered by Edelkamp (e.g., 2001), developed further by Haslum, Bonet, and Geffner (2005), and the current state of the art is set by the "incremental PDB" (iPDB) procedure by Haslum et al. (2007).

However, unlike the situation in other application areas of heuristic search, PDB heuristics have lagged behind the state of the art in planning. For example, none of the planning systems that won the International Planning Competitions (IPCs) have made use of PDBs, and *merge-and-shrink heuristics* (Helmert, Haslum, and Hoffmann 2007; Nissim, Hoffmann, and Helmert 2011a) have been the best-performing planning heuristics based on (homomorphic) *abstraction*, the class of heuristic approaches into which PDBs fall (Helmert and Domshlak 2009).

One significant challenge when applying PDB heuristics to planning, as opposed to classical search problems, is that PDB construction time must be amortized within a single planner run: each planning task that a planner is evaluated on has its own state space, set of actions and goal, which makes it impossible to compute a PDB once and then reuse it for multiple inputs. Hence, it is critically important to compute pattern databases *efficiently* – even more so when using PDBs within an overall algorithm like iPDB, which generates hundreds or even thousands of individual pattern databases in order to find a good pattern collection.

In this paper, we describe such an efficient PDB implementation and show that a planner leveraging this implementation outperforms previous PDB-based planners and is competitive with the strongest merge-and-shrink heuristics.

Background

We assume that the reader is familiar with the basics of pattern databases, state spaces and planning tasks, and hence we only revise the necessary background briefly.

To build a PDB for a planning task Π, we need a description of Π along with the set of state variables that shall define the pattern. We assume that planning tasks are given in the SAS$^+$ representation augmented with operator costs, as in the work of Haslum et al. (2007).

Such a planning task is defined over a set of *state variables* $\mathcal{V}$, where each variable $v \in \mathcal{V}$ has an associated *finite domain* $\mathcal{D}_v$. A *partial variable assignment* consists of a set of pairs of the form $\langle v, d\rangle$ with $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$ such that all variables in the set are different. A *state* is a partial variable assignment containing such a pair for each state variable in $\mathcal{V}$. The number of states is thus $\prod_{v\in\mathcal{V}} |\mathcal{D}_v|$. For state s (or more generally: for partial variable assignment s), we write $s[v]$ for the value associated with variable $v \in \mathcal{V}$.

A planning task has a *goal*, which is a partial variable assignment $s_\star$, with the semantics that every state s with $s_\star \subseteq s$ is a goal state. It also has a set of *operators* $\mathcal{O}$, where each operator o has a *precondition* $pre(o)$ and *effect* $eff(o)$, which are partial variable assignments that define in which states an operator can be applied (all s with $pre(o) \subseteq s$) and what the resulting state is (the state s' that agrees with $eff(o)$ on all variables mentioned in the effect and with s on all other variables; we write this state as $app_o(s)$). Operators also have an associated *cost* $cost(o) \in \mathbb{N}_0$.

The *goal distance* $h^*(s)$ of a state s is the cost of a shortest path from s to any goal state in the state space defined by the states and operators of the task, where state transitions are weighted by the cost of the operators that induce them. (If no goal state is reachable from s, we have $h^*(s) = \infty$.)

Pattern databases are defined by a subset of variables $P \subseteq \mathcal{V}$ called the *pattern*. They are based on an abstraction of the state space of Π where states which agree on all variables in P are considered equivalent. In the case of SAS$^+$ tasks, the abstract state space induced by a pattern database heuristic is identical to the state space of the

planning task obtained by syntactically removing all references to state variables that are not part of the pattern from all parts of the task description. Therefore, the abstract state space for a given pattern is itself the state space of a planning task. (Note that this "syntactic abstraction" property does *not* hold for some more general planning formalisms, such as ones supporting axioms or preconditions using arbitrary propositional logic formulae.)

Pattern Databases for Planning

Since the abstract state spaces that define PDB heuristics are themselves described by planning tasks, the first step in computing a PDB is to compute a representation of this abstract planning task, which is a straightforward operation. Since the resulting abstract planning task is just a regular planning task and we will not need to refer to the concrete planning task from which the abstract planning task is derived, we use the notations introduced earlier ($\mathcal{V}$, $\mathcal{O}$, $s_\star$, etc.) to refer to the components of this abstract task in the following. We assume that the variables are numbered as $\mathcal{V} = \{v_1, \ldots, v_k\}$ and write $\mathcal{D}_i$ as an abbreviation for $\mathcal{D}_{v_i}$.

A pattern database is a look-up table which contains the value $h^*(s)$ for every state of the (abstract) task. It is usually implemented as a one-dimensional, zero-indexed array of size $N := \prod_{i=1}^{k} |\mathcal{D}_i|$. To look up the value for state s in this table, we use a perfect hash function from states to indices in $\{0, \ldots, N-1\}$. Such indices are called *ranks*, the process of computing ranks from states is called *ranking*, and the inverse process is called *unranking*. For simplicity, we assume that the domains of the variables are 0-indexed integers (i.e., $\mathcal{D}_i = \{0, 1, \ldots, |\mathcal{D}_i| - 1\}$ for all variables v_i), in which case a ranking function is given by

$$\mathit{rank}(s) = \sum_{i=1}^{k} N_i s[v_i],$$

where the coefficients $N_i := \prod_{j=0}^{i-1} |\mathcal{D}_j|$ can be precomputed. The corresponding unranking function is given by

$$\mathit{unrank}(r)[v_i] = \left\lfloor \frac{r}{N_i} \right\rfloor \bmod |\mathcal{D}_i|.$$

Ranking and unranking can be performed in time $O(k)$, and the value of a given state variable in a ranked state can be computed in $O(1)$ (i.e., it is not necessary to unrank a state if we only want to access the value of one or a few variables).

In the following, we describe two algorithms for computing pattern database heuristics in planning. The first algorithm, the "basic" one, is straightforward and serves as a basis for the second one, the "efficient" one. Both algorithms compute the same pattern database, but differ in runtime and memory requirements.

Basic PDB Construction Algorithm

Efficient procedures for computing pattern databases perform a search of the state space in the backward direction, starting from the goal states of the task and searching towards all other states. This allows computing the goal distances of every state in a single sweep, for example using

```
   Input: v_1, ..., v_k: state variables (D_i: domain of v_i)
          s_⋆: goal
          O: operators
 1 N := ∏_{i=1}^{k} |D_i|
 2 PDB := array of size N filled with ∞
 3 heap := make-heap()
 4 graph := make-array-of-vectors()
   /* phase 1: create graph of backward transitions and
      identify goal states                                  */
 5 for r ∈ {0, ..., N − 1} do
 6     s := unrank(r)
 7     if s_⋆ ⊆ s then
 8         PDB[r] := 0
 9         heap.push(0, r)
10     end
11     for o ∈ O do
12         if pre(o) ⊆ s then
13             s' := app_o(s)
14             r' := rank(s')
15             graph[r'].append(⟨r, cost(o)⟩)
16         end
17     end
18 end
   /* phase 2: perform Dijkstra search with graph and
      heap to complete the entries in PDB                   */
19 ... (Dijkstra pseudo-code omitted)
```

Algorithm 1: Basic PDB construction algorithm. In our implementation, in line 6 we do not actually unrank r from scratch, but use an incremental algorithm to create s based on the state from the previous iteration, which takes amortized $O(1)$ time instead of $O(k)$.

Dijkstra's algorithm (Dijkstra 1959). For the combinatorial puzzles which are typically studied in the heuristic search literature (e.g., Yang et al. 2008), such a backward search is a straightforward operation since operators are invertible. In planning, however, this is in general not the case, as operators need not be injective. For example, an operator with $\mathit{pre}(o) = \{v_1 \mapsto 3\}$ and $\mathit{eff}(o) = \{v_2 \mapsto 1, v_3 \mapsto 0\}$ can reach the state $\{v_1 \mapsto 3, v_2 \mapsto 1, v_3 \mapsto 0\}$ from many different states, namely all states s where $s[v_1] = 3$, independently of $s[v_2]$ and $s[v_3]$.

In the basic PDB construction algorithm, we work around this issue by performing the PDB computation in two phases: in the first phase we explicitly construct a directed graph representing the state space of Π by iterating over all states s and operators o, testing whether o is applicable in s, and if so, inserting a (backward) edge from $\mathit{app}_o(s)$ to s with cost $\mathit{cost}(o)$. In the second phase, after the graph is constructed, it is searched with Dijkstra's algorithm. The advantage of this approach is that it is one of the fastest ways (in terms of big-O notation) to compute the pattern database, since (potentially complicated) reasoning about backward reachability is avoided. The disadvantage is that a represen-

tation of all transitions needs to be kept in memory. To keep this representation small, we store the *ranks* of states rather than the states themselves in all data structures. In a straightforward implementation, this requires 8 bytes per transition (4 for the rank of the state reached by the transition, plus 4 for the cost of the inducing operator). Complete pseudocode for the basic PDB construction algorithm is given in Algorithm 1.

Efficient PDB Construction Algorithm

The basic PDB construction algorithm has three major inefficiencies:

1. Creating the complete transition graph a priori has a significant space cost.
2. Testing each operator for applicability in each state (i.e., the nested loops over r and o in Algorithm 1) is expensive. In most planning tasks only a small fraction of operators is applicable in each state.
3. The algorithm must compute and rank many states in lines 13 and 14. A complexity analysis shows that these computations can form the main bottleneck of the overall algorithm in terms of big-O performance. A suitable implementation of Dijkstra's algorithm runs in time $O(N \log N + T)$ where N is the number of states and T the number of state transitions.[1] Transition systems in planning are often very dense, so that T is much larger than $N \log N$. In this case, the combined cost of all the state generations and ranking in lines 13 and 14, which is $O(T \cdot k)$, dominates overall runtime.

In the remaining three subsections, we discuss how our efficient PDB implementation addresses these inefficiencies.

Avoiding Constructing the Transition Graph

To avoid constructing transition graphs a priori, we must be able to efficiently *regress* over states, i.e., given a state s', find all state/operator pairs $\langle s, o \rangle$ such that $app_o(s) = s'$. Given such a regression operation, we can generate the necessary predecessor edges and their edge costs on the fly whenever we expand a state in Dijkstra's algorithm.

The problem with regression, as hinted before, is that planning operators are not generally injective, and hence s' and o do not uniquely define the predecessor state s. Fortunately, there is a well-known compilation that addresses this issue: whenever an operator o mentions variable v in its effect but not in its precondition, o is noninjective. However, it can be transformed into a set of injective operators by generating one copy of the operator for each possible value of v (repeating the process if there are further variables mentioned in the effect but not precondition). For example, if $\mathit{pre}(o) = \{v_2 \mapsto 3\}$, $\mathit{eff}(o) = \{v_1 \mapsto 1, v_2 \mapsto 1\}$ and $\mathcal{D}_1 = \{0, 1, 2\}$, then we create three versions o^0, o^1, o^2 of the operator, where $\mathit{eff}(o^i) = \mathit{eff}(o)$ and $\mathit{pre}(o^i) = \mathit{pre}(o) \cup \{v_1 \mapsto i\}$.

For general planning, this problem transformation is somewhat dangerous because it can exponentially blow up the representation size: consider an operator o with empty precondition which sets n variables to 0 in its effect. If all variables have the domain $\{0, 1\}$ the compilation gives rise to 2^n versions of the operator, which is unacceptable in general. However, in the context of pattern databases, this increase is not an issue since we perform a complete explicit search on the state space of the (abstract) problem anyway, and hence the 2^n edges in the state space induced by o would be considered in any case. The transformation only affects the number of operators in the problem, but *not* the number of edges in the transition graph, and hence it does not increase the asymptotic cost of PDB generation.

Therefore, in our efficient PDB generation algorithm we "multiply out" all noninjective operators in this fashion before starting PDB construction. This allows us to regress operators easily: in state s' we can regress over o iff $\mathit{eff}(o) \subseteq s'$ and $s'[v] = \mathit{pre}(o)[v]$ for all variables v mentioned in $\mathit{pre}(o)$ but not $\mathit{eff}(o)$. The resulting state s of regressing s' over o is the state that agrees with $\mathit{pre}(o)$ on all variables mentioned in $\mathit{pre}(o)$ and with s' on all other variables. This is equivalent to a regular *progression* (forward application) in s' of an operator $\tilde{o}$ defined as

$$\begin{aligned} \mathit{pre}(\tilde{o}) &:= \mathit{eff}(o) \cup \\ &\qquad \{\langle v, d \rangle \in \mathit{pre}(o) \mid v \text{ does not occur in } \mathit{eff}(o)\} \\ \mathit{eff}(\tilde{o}) &:= \mathit{pre}(o), \text{ and} \\ \mathit{cost}(\tilde{o}) &:= \mathit{cost}(o). \end{aligned}$$

Avoiding Checking All Operators Individually

In the previous step, we have transformed the problem of regressing state s' over all operators $\mathcal{O}$ into the problem of *progressing* s' over all operators $\{\tilde{o} \mid o \in \mathcal{O}\}$. Hence, to address the second inefficiency we need to efficiently determine all operators $\tilde{o}$ applicable in s' without checking each of them individually.

Fortunately, a suitable data structure for this purpose already exists in the form of *successor generators* used in the Fast Downward planner (Helmert 2006, Section 5.3.1). The technical details of successor generators are somewhat involved and need not be repeated here. For our purposes, it is sufficient to know that they can be constructed in time linear in the total representation size of all operators (i.e., $O(\sum_{o \in \mathcal{O}}(|\mathit{pre}(\tilde{o})| + |\mathit{eff}(\tilde{o})|))$) and, once constructed, allow computing all applicable operators in a state s' much more efficiently than by testing each operator individually (usually in sublinear time in $|\mathcal{O}|$, and linear in the number of operators applicable in s in the best case).

Therefore, our improved PDB construction algorithm uses successor generators for the operators $\tilde{o}$ for an efficient on-the-fly computation of the predecessor states of any state s' expanded during Dijkstra's algorithm.

[1] Our implementation uses delayed duplicate elimination and has an inferior worst-case performance of $O(N \log N + T \log N)$. Despite the big-O disadvantage, we believe our implementation to be preferable for most planning benchmarks since the hidden constants are lower and the worst case can only arise if operators have wildly varying cost. In particular, our algorithm also runs in $O(N \log N + T)$ if operator costs are bounded by a constant.

Avoiding State Generation and Ranking

The final change we make to the basic algorithm in order to make it more efficient is to avoid ranking and unranking of states while running Dijkstra's algorithm. Instead, all computations are directly performed on the ranks r' that form the indices into the PDB.

Our first observation is that we can easily modify the successor generator introduced in the previous step to not require an unranked state. Instead, whenever the successor generator needs the value of a state variable v_i in the state represented by r', we can compute $\mathit{unrank}(r')[v_i]$ on the fly in time $O(1)$, as discussed earlier when we introduced the *rank* and *unrank* functions. This allows us to directly compute the set of operators $\tilde{o}$ which are applicable in the state represented by r'.

The remaining piece of the puzzle is to find a way to *apply* $\tilde{o}$ to the state with rank r' without performing any ranking or unranking. Given r' and $\tilde{o}$, we are interested in the rank of the state r that is reached when applying $\tilde{o}$ in the state represented by r'. That is, we want to compute $r := \mathit{rank}(\mathit{app}_{\tilde{o}}(\mathit{unrank}(r')))$, and we want to do this without explicitly ranking or unranking.

This is easy to do in the case where $\mathit{pre}(\tilde{o})$ and $\mathit{eff}(\tilde{o})$ mention exactly the same set of variables: in this case it is possible to precompute a number $\Delta(o)$ such that for all ranks r', $\mathit{rank}(\mathit{app}_{\tilde{o}}(\mathit{unrank}(r'))) = r' + \Delta(\tilde{o})$. In the general case where the precondition and effect refer to different variables, however, such a number $\Delta(\tilde{o})$ does not exist: if applying $\tilde{o}$ in r'_1 results in a state with rank r_1 and applying $\tilde{o}$ in r'_2 results in a state with rank r_2, then there is no general guarantee that $r_1 - r'_1 = r_2 - r'_2$. However, we can avoid this issue by making sure that in each operator the variables mentioned in the precondition and in the effect are always the same, using the same kind of transformation as in our first optimization (but this time applied to state variables mentioned in $\mathit{pre}(o)$ but not $\mathit{eff}(o)$, rather than vice versa).

After this transformation, we can apply operators directly to ranked states by simply adding $\Delta(\tilde{o})$ to the rank in time $O(1)$, improving over the $O(k)$ successor generation in the basic PDB generation algorithm.

Experimental Results

To evaluate our PDB construction algorithm, we implemented it within Fast Downward, a state-of-the-art planning system (Helmert 2006). We performed two sets of experiments. The first set of experiments is a comparison "in the small" that evaluates the PDB construction algorithm itself as an isolated component. The main question we want to answer with this set of experiments is:

1. Do the modifications to the basic PDB construction algorithm discussed in the previous section lead to improved performance in terms of runtime and memory usage? If so, to what extent?

The second experiment is a comparison "in the large" that evaluates the new PDB construction algorithms within the context of an overall PDB-based planning system. The main question we want to answer with this set of experiments is:

2. Is a PDB-based planning system using our PDB construction algorithm competitive with other abstraction-based approaches to optimal planning? If yes, to what extent can planner performance be attributed to efficient implementation techniques rather than general properties of PDB-based heuristics?

All experiments were conducted on dual-CPU Opteron 2384 machines, on which we ran eight experiments simultaneously in order to fully utilize the eight available cores. Each individual experiment ran on a single core with a time limit of 30 minutes and a memory limit of 2 GB. As a benchmark set, we used the planning tasks from the sequential optimization track of the last International Planning Competition (IPC 2011). This benchmark set comprises a total of 280 planning tasks from 14 different planning domains.

Comparison of PDB Construction Algorithms

In the first set of experiments, each experiment consisted in the construction of a single PDB with a prespecified size limit (in terms of number of abstract states), using either the basic PDB construction algorithm or efficient PDB construction algorithm. No search was performed after constructing the PDB, as we were only interested in the PDB construction algorithm itself here.[2]

We first show the general *coverage* of both implementations, as the basic implementation often fails to construct a PDB within the given time and memory limits. Table 1 shows the number of planning tasks in each IPC 2011 benchmark domain for which a PDB could be computed for three different PDB size limits.

The experiment shows that the efficient implementation is able to complete PDB construction in significantly more cases, especially for the PDB size limit settings of 10,000,000 and 100,000,000. For the latter limit, the efficient algorithm managed to compute a PDB in 262 out of 280 cases, while the basic algorithm only succeeded in 30 cases. When the basic implementation failed to construct a PDB, this was most often because it exhausted the 2 GB memory limit.

To give a more fine-grained picture of the performance of the two construction algorithms, Table 2 presents detailed experiments for 2 of the 14 domains, comparing time and memory needed to generate a PDB. We restrict the comparison to the smallest size limit of 100,000, as the basic implementation often fails to construct a PDB for larger values.

The results show that the efficient implementation is *much faster* and requires *much less memory* to construct PDBs. Consequently, the efficient implementation succeeds

[2]Note that a size limit of N means that the PDB has *at most* N entries, but the actual number can be significantly lower. For example, for $N = 100000$ and variables which all have a domain with 12 elements, the pattern can only consist of 4 variables because $12^4 = 20736 \leq 100000$, but $12^5 > 100000$. The procedure we used to define the pattern in this experiment adds the variables to the pattern in the order used by the original merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007), stopping when the next variable in the sequence would increase the PDB size beyond N.

Domain	basic algorithm				efficient algorithm			
	100k	**1m**	**10m**	**100m**	**100k**	**1m**	**10m**	**100m**
Barman (20)	**20**	**20**	0	0	**20**	**20**	**20**	**20**
Elevators (20)	**20**	**20**	18	0	**20**	**20**	**20**	**20**
Floortile (20)	**20**	**20**	2	0	**20**	**20**	**20**	**20**
Nomystery (20)	**20**	**20**	18	10	**20**	**20**	**20**	**20**
Openstacks (20)	**20**	**20**	3	0	**20**	**20**	**20**	**20**
Parcprinter (20)	**20**	**20**	4	0	**20**	**20**	**20**	**20**
Parking (20)	**20**	**20**	10	0	**20**	**20**	**20**	**20**
Pegsol (20)	**20**	**20**	0	0	**20**	**20**	**20**	**20**
Scanalyzer (20)	17	12	3	3	**20**	**20**	**19**	**18**
Sokoban (20)	**20**	**20**	**20**	7	**20**	**20**	**20**	**20**
Tidybot (20)	13	0	0	0	**20**	**20**	**20**	**4**
Transport (20)	**20**	**20**	18	2	**20**	**20**	**20**	**20**
Visitall (20)	**20**	**20**	8	8	**20**	**20**	**20**	**20**
Woodworking (20)	**20**	**20**	2	0	**20**	**20**	**20**	**20**
Total (280)	270	252	106	30	**280**	**280**	**279**	**262**

Table 1: Comparison of number of instances where a PDB could be constructed within the given limits by the basic and efficient construction algorithm. Number of tasks in each domain is shown in parentheses. The PDB size limits are abbreviated as **100k** for 100,000; **1m** for 1,000,000; **10m** for 10,000,000; and **100m** for 100,000,000. Best results for each size limit are highlighted in bold.

in building a PDB for many problem instances where the slow implementation fails.

Comparison of Full Planners

In the second set of experiments, we investigated if the efficient PDB construction algorithm can help build a PDB-based planning system that is competitive with the state of the art in optimal planning. In particular, we are interested in comparisons to the *previous best PDB-based planning systems* and to *merge-and-shrink* heuristics (Helmert, Haslum, and Hoffmann 2007; Nissim, Hoffmann, and Helmert 2011a). Merge-and-shrink heuristics are a related approach to pattern database heuristics and were among the most successful approaches at IPC 2011. (The only planners that outperformed the merge-and-shrink planner by Nissim, Hoffmann and Helmert at IPC 2011 were portfolio planners that used merge-and-shrink as a component.)

Before our work, the state of the art for PDB-based planning systems was set by the implementation of the iPDB procedure (Haslum et al. 2007) within Haslum's HSP_f planner. Previous experiments (Helmert, Haslum, and Hoffmann 2007) have shown a planning system using merge-and-shrink heuristics to clearly outperform this iPDB implementation. However, this result is difficult to interpret since the merge-and-shrink and iPDB approaches were implemented within completely different planning systems, and implementation details can be very important for performance.

The iPDB approach is based on a local search in the space of PDB collections and constructs a large number of (typically small to medium-sized) pattern databases before starting the actual search. We hoped that an iPDB implementation using our efficient PDB construction could outperform

I	basic (100k) Mem.	basic (100k) Time	eff. (100k) Mem.	eff. (100k) Time	I	basic (100k) Mem.	basic (100k) Time	eff. (100k) Mem.	eff. (100k) Time
Scanalyzer					Scanalyzer				
01	5	0.02	**5**	**0.01**	11	862	21.86	**24**	**0.41**
02	43	0.97	**6**	**0.15**	12	18	0.25	**6**	**0.06**
03	43	0.66	**6**	**0.11**	13	50	2.71	**19**	**0.05**
04	66	1.31	**6**	**0.05**	14	62	4.52	**21**	**0.05**
05	125	2.53	**7**	**0.07**	15	32	0.63	**6**	**0.03**
06	28	0.45	**6**	**0.04**	16	217	2.78	**8**	**0.17**
07	—	—	**27**	**4.74**	17	126	2.78	**10**	**0.07**
08	1156	31.60	**10**	**0.76**	18	727	20.15	**14**	**0.46**
09	834	19.63	**13**	**0.48**	19	930	24.13	**18**	**0.50**
10	—	—	**18**	**3.03**	20	—	—	**236**	**30.45**
Tidybot					Tidybot				
01	437	8.49	**11**	**0.42**	11	1574	28.87	**34**	**1.27**
02	562	11.16	**16**	**0.51**	12	1573	29.03	**34**	**1.36**
03	562	13.28	**16**	**0.51**	13	1573	28.49	**34**	**1.14**
04	562	11.75	**16**	**0.53**	14	—	—	**52**	**1.57**
05	1184	20.87	**23**	**0.87**	15	—	—	**52**	**1.40**
06	1184	22.49	**23**	**0.87**	16	—	—	**53**	**1.46**
07	1184	24.38	**23**	**0.88**	17	—	—	**53**	**1.43**
08	1184	20.74	**23**	**0.88**	18	—	—	**52**	**1.86**
09	1576	29.04	**34**	**1.10**	19	—	—	**53**	**1.42**
10	1573	27.95	**34**	**1.11**	20	—	—	**53**	**1.47**

Table 2: Comparison of peak memory usage (in MB) and construction time (in seconds) for the basic and efficient construction algorithm with a PDB size limit of 100,000. The first column indicates the instance of the domain. Best results are highlighted in bold.

the implementation in HSP_f and possibly approach the performance of merge-and-shrink-based planners.

We reimplemented the iPDB approach within the Fast Downward planner, which is also the basis of the merge-and-shrink heuristic results reported in the literature. Our reimplementation performs the same local search in pattern space as described in the original iPDB paper (Haslum et al. 2007), with very few *conceptual* differences.[3] The most important such differences are the following:

- The iPDB pattern construction process requires heuristic estimates for a set of sample states and a large number of candidate patterns. These candidate patterns are of the form $P \cup \{v\}$ where P is a pattern for which a PDB has previously been constructed and $v \notin P$ is an additional variable considered for adding.

  Our implementation computes a complete PDB for $P \cup \{v\}$ to obtain these heuristic values. The HSP_f implementation uses a more sophisticated strategy where $h_{P \cup \{v\}}(s)$ is computed using A* search with h_P as a heuristic. We avoided this additional complexity because we hoped that

[3] Some differences are hard to avoid since the two underlying planning systems are very different and the iPDB procedure is complex. To give some indication of this complexity, the PDB-related code within HSP_f comprises around 7000 lines, and the PDB-related code within Fast Downward comprises around 2000 lines. The complete planning systems comprise more than 100000 lines of code in the case of HSP_f and more than 30000 in the case of Fast Downward, although in both cases not of all this code is relevant to the experiments here.

our PDB construction process was sufficiently fast for computation of $h_{P\cup\{v\}}$ not to be a major bottleneck.

- The HSP_f implementation of iPDB uses a statistical pruning technique for reducing the amount of samples required for finding the next pattern to add to the pattern collection. To reduce code complexity, we did not implement this pruning criterion, as we hoped it would not be necessary for good performance in our implementation.
- The PDBs used by HSP_f are *constrained pattern databases* (Haslum, Bonet, and Geffner 2005), which can offer better heuristic estimates than regular pattern databases. To reduce code complexity, we did not implement constrained PDBs.

Besides these conceptual differences, which should favor HSP_f, there are significant *engineering differences* between the two implementations, mostly related to the choice of data structures that admit efficient algorithms. Our aim was to provide an iPDB implementation that is optimized to a similar degree as the other heuristics within Fast Downward.

Of course, one such engineering difference is that our implementation uses the efficient PDB construction algorithm described in this paper.[4] Apart from the addition of pattern database heuristics and the iPDB procedure, which are described in detail by Haslum et al. (2007), we did not make any changes to the basic Fast Downward planner, so we refer to the literature for more details on the planner, such as the method it uses to convert a PDDL task into a finite-domain representation (Helmert 2006; 2009).

We evaluated our implementation of iPDB by comparing it to the original iPDB implementation in Haslum's HSP_f planner (in its current version) and to M&S-2011, the merge-and-shrink based planning system that participated in IPC 2011 (Nissim, Hoffmann, and Helmert 2011a; 2011b).

We performed some initial parameter tuning for the two iPDB approaches, resulting in the following settings, which were used for all benchmark instances:

- *max. pattern size* $2 \cdot 10^6$*:* no PDBs with more than 2 million entries are considered in the pattern construction phase
- *max. pattern collection size* $2 \cdot 10^7$*:* the sum of PDB sizes for the final pattern collection may not exceed 20 million
- *sample size 100:* each step of the local search in pattern space samples 100 states to determine the next pattern to add to the collection
- *min. improvement 10:* if fewer than 10 of the sampled states lead to an improvement of heuristic estimates, the pattern generation process stops and the actual search for a solution of the planning task commences

Table 3 shows the outcome of this experiment. It reports the number of problem instances solved by each planner in each domain within the time and memory limits. Our new iPDB implementation clearly outperforms the earlier iPDB implementation and achieves the same overall coverage as M&S-2011. We point out that M&S-2011 is a portfolio of two different planners, each of which is run for part of the 30 minute timeout with no communication between them. The new iPDB implementation beats each of the two component planners that form this portfolio handily when these are considered in isolation (they achieve a coverage of 84 and 131), and a portfolio including both FD-iPDB and the components of M&S-2011 would result in better coverage than any of the planners reported here.

This clearly shows that PDB-based planners can be competitive with the state of the art in optimal planning, but also that an efficient implementation is necessary to achieve this.

Domain	**HSP_f-iPDB**	**FD-iPDB**	**M&S-2011**
Barman (20)	**4**	**4**	**4**
Elevators (20)	**19**	15	10
Floortile (20)	6	2	**7**
Nomystery (20)	**18**	16	**18**
Openstacks (20)	6	**14**	13
Parcprinter (20)	**13**	11	**13**
Parking (20)	**5**	**5**	**5**
Pegsol (20)	5	18	**19**
Scanalyzer (20)	7	**10**	9
Sokoban (20)	15	**20**	19
Tidybot (20)	**14**	**14**	7
Transport (20)	**7**	6	**7**
Visitall (20)	**16**	**16**	**16**
Woodworking (20)	6	5	**9**
Total (280)	141	**156**	**156**

Table 3: Comparison of solved tasks for the original iPDB implementation in HSP_f (HSP_f-iPDB), our new implementation of iPDB in Fast Downward (FD-iPDB) and the IPC 2011 merge-and-shrink planner (M&S-2011). Number of tasks in each domain is shown in parentheses. Best results are highlighted in bold.

Conclusion

We have described and experimentally evaluated an efficient implementation of pattern database heuristics for optimal planning. Our experiments show significant efficiency improvements over a basic PDB generation algorithm and establish that a planning algorithm based on the efficient PDB generation algorithm is competitive with the state of the art in optimal planning.

Somewhat surprisingly for us, the resulting planning system not just outperforms previous PDB-based planning systems, but also reaches the performance of a state-of-the-art portfolio planner based on merge-and-shrink heuristic. This does not imply that PDB heuristics are stronger than merge-and-shrink heuristics in general: while the merge-and-shrink planners in the portfolio use A^* with a single heuristic, the iPDB procedure uses a sophisticated strategy to select and combine a large number of pattern databases. We see this result as further evidence that intelligently *combining* heuristics is an important issue in optimal planning that requires further research and promises significant performance improvements in the future.

[4] The PDB construction algorithm in HSP_f is somewhere between our basic and efficient algorithms. Like our efficient algorithm it avoids constructing an explicit graph. However, it does not attempt to minimize ranking and unranking operations.

Acknowledgments

We would like to thank Patrik Haslum for patiently answering our numerous questions about the iPDB implementation in HSP_f and for his help in conducting the experiments for this paper.

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) 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. 1996. Searching with pattern databases. In *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-96)*, volume 1081 of *Lecture Notes in Artificial Intelligence*, 402–416. Springer-Verlag.

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

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

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.

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., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. 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.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011a. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1983–1990.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011b. The Merge-and-Shrink planner: Bisimulation-based abstraction for optimal planning. In *IPC 2011 planner abstracts*, 106–107.

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