Partial Pattern Databases

Kenneth Anderson, Robert Holte, and Jonathan Schaeffer

University of Alberta, Computing Science Department,
Edmonton, Alberta, T6G 2E8, Canada
{anderson, holte, jonathan}@cs.ualberta.ca

Abstract. Perimeters and pattern databases are two similar memory-based techniques used in single-agent search problems. We present partial pattern databases, which unify the two approaches into a single memory-based heuristic table. Our approach allows the use of any abstraction level. We achieve a three-fold reduction in the average number of nodes generated on the 13-pancake puzzle and a 27% reduction on the 15-puzzle.

1 Introduction

Perimeters and pattern databases (PDBs) are two successful techniques that improve forward search in single-agent search problems. They have proven effective at improving search performance when combined with minimal-memory, depth-first search techniques such as IDA* [12]. Pattern databases, in particular, have been used to great effect in solving puzzle, DNA sequence alignment, and planning problems [2, 18, 4].

Perimeters and pattern databases are very similar in their approach to speeding up search. Both techniques use retrograde (reverse) search to fill a memory-based heuristic table. However, pattern databases use abstraction when filling this table, whereas perimeters use none. Also, the memory limit determines the abstraction level for pattern databases; the full PDB must completely fit in memory. Perimeters on the other hand, are built without any abstraction; the perimeter stops being expanded when a memory limit is reached.

We present two general techniques that allow the use of arbitrary abstraction levels in between the two extremes. *Partial pattern databases* use memory similarly to the perimeters, storing part of the space in a hash table. *Compressed partial PDBs* use memory more efficiently, like pattern databases. Our unifying approach allows flexibility when choosing the abstraction level. Through testing, we can determine the best abstraction level for our domains.

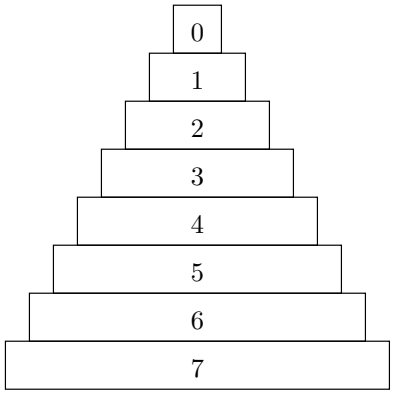
We test on two complimentary puzzle domains: the K-pancake puzzle, which has a large branching factor, and the 15-puzzle, which has a small branching factor. Our techniques are compared against full pattern databases, which have proven very effective on these domains. On the 13-pancake puzzle, keeping memory constant, we reduce the average number of generated nodes by a factor of three. On the 15-puzzle, keeping memory constant, we reduce the average number of nodes generated by 27%.

Section 2 examines related work on perimeters, pattern databases, and previous attempts at combining the two approaches. Partial pattern databases and compressed partial PDBs are introduced in Section 3. Results on the K-pancake puzzle and 15-puzzle are shown in Section 4 and Section 5, respectively. Section 6 presents our final analysis and possible extensions to our approach.

2 Background

The domains used in this paper are the K-pancake puzzle and the 15-puzzle. The K-pancake puzzle consists of a stack of K pancakes all of different sizes, numbered 0 to $K-1$ (Figure 1). There are $K-1$ operators, where operator k ($1 \leq k \leq K-1$) reverses the order of the top k pancakes. We refer to an individual pancake as a *tile* and its placement in the stack as a *location*.

The 15-puzzle is comprised of a 4 by 4 grid of tiles, with one location empty. The empty location is called the *blank*. Valid operations include swapping the blank with one of up to 4 adjacent tiles. Figure 1 shows a possible arrangement of tiles for the 15-puzzle.



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig. 1. Goal node for 8-pancake (left) and 15-puzzle (right).

A *node* is a unique arrangement of tiles. Given a *start* and *goal* node, we wish to find a path (series of operators) of minimal cost leading to the goal node. In our domains, operators have unit cost. However, the ideas in this paper are applicable to any domain that can be structured as a graph with directed, weighted edges of non-negative cost.

IDA* is a traditional search technique proven to work well in these two domains. IDA* is a depth-first search method that finds optimal solution paths through a directed weighted graph [12]. Nodes are pruned if the total cost through a node, $f(n)$, exceeds a bound f_{limit} according to the definition $f(n) =$

$g(n)+h(n)$, where $g(n)$ is the actual cost to node n and h is the heuristic estimate of the remaining cost to the goal. IDA* iteratively increases the bound, f_{limit}, and re-searches until the goal is found. If h is *admissible*, does not overestimate the cost to the goal, then IDA* guarantees finding an optimal path, provided that one exists.

If a heuristic is *consistent*, then the change in heuristic between any two nodes in the graph is less than or equal to the cost between the two nodes. Consistency implies admissibility. If IDA* is used with an inconsistent heuristic, heuristic values can be improved through the use of bidirectional-pathmax (BPMX) [8]. BPMX propagates heuristic inconsistencies through the search tree from parent-to-child and child-to-parent, increasing heuristic values when appropriate. This technique uses no extra memory in IDA* and takes very little time.

The *original space*, S, consists of the set of nodes that can reach the goal through a series of operators. An *abstract space* is a set of abstract nodes and all applicable operators, where every node in the original space maps to some corresponding node in the abstract space. We use *domain abstraction* as our abstraction technique [10]. The abstraction is created by renaming specific tiles to the same name, x. These re-named tiles are called *don't-care* tiles while the other tiles are called *unique tiles*. In the case of the 15-puzzle, the blank will always be a unique tile. When we refer to *abstraction-N*, we refer to a specific abstraction with N unique tiles (the actual tiles vary with the domain). A *coarse-grained* abstraction has fewer unique tiles (and hence covers more abstract nodes) than a *fine-grained* abstraction.

The following are memory-based heuristic methods. Both techniques use *retrograde* (backwards) search from the goal to improve or create heuristic values, but they achieve this in quite different ways.

2.1 Perimeters

Perimeter search is a type of bidirectional search technique and requires a predecessor function. Originally proposed by Dillenburg and Nelson, perimeter search performs two successive searches [3]. The first search proceeds in the backwards direction from the goal, forming a set of perimeter nodes P, which encompass the goal. A node n is *on* the perimeter if $n \in P$, *inside* the perimeter if it was expanded during perimeter creation, and *outside* the perimeter if it was not expanded. Any node outside the perimeter must pass through some node in the perimeter to reach the goal. Many techniques can be applied to generate the perimeter: Breadth-first search creates a *constant-depth perimeter* [3, 14]; A* creates a *constant-evaluation perimeter* [3]; and expansion based on heuristic difference creates another kind of perimeter [11].

If the perimeter is generated for one problem instance, then the backward and forward searches are performed in series. The backward search forms a perimeter and, if the start node is not found, it is followed by the forward search. However, if the perimeter is constructed for use on multiple problem instances with the same goal, then the interior of the perimeter, set A, is stored. When the forward search begins, if the start is in set A, the actual cost to the goal is known so the

heuristic is corrected to this value. Otherwise the start is outside the perimeter and the forward search begins.

The second, forward search progresses either from the start to the perimeter, called *front-to-front* evaluation, or from the start to the goal, called *front-to-goal* evaluation. Front-to-front evaluation calculates the heuristic value of a node based on the estimated cost through every node on the perimeter. Although larger perimeters provide better heuristic values, the heuristic takes increasingly longer to compute. Additionally, front-to-front evaluation requires a heuristic to exist between any two distinct nodes.

By contrast, search using front-to-goal evaluation requires only a heuristic to the goal. The heuristic of nodes found to be inside the perimeter is corrected using the exact cost to the goal. The heuristic of nodes outside the perimeter can sometimes be corrected; here are two different approaches for correcting the heuristic values using front-to-goal evaluation. Using a depth-limited perimeter where d is the cost bound, d is a lower bound on the true cost to the goal for all nodes outside the perimeter [1]. Or, given a consistent heuristic, a correction factor is equal to the lowest difference between the actual cost to the goal and the estimated heuristic cost to the goal [11]. The correction factor is now added to the original heuristic if outside the perimeter.

Any search technique will work for the forward search, but IDA* and similar low-memory search techniques are most commonly employed for their adaptability, scalability, and low memory requirements [3, 14, 11]. We use IDA* throughout this paper.

2.2 Pattern Databases

Introduced by Culberson and Schaeffer, pattern databases use abstraction to create a heuristic lookup table [2]. Retrograde search starts from the abstract goal. The search proceeds backwards applying all applicable reverse operators until the space is covered. The costs from the abstract goal in the abstract space are recorded in a table and used as a heuristic in the forward search. This produces an admissible and consistent heuristic.

Holte *et al.* have investigated generating and caching parts of pattern databases during search in [9]. Similarly, Zhou and Hansen have demonstrated a technique whereby provably unnecessary parts of the PDB are not generated (given an initial consistent heuristic and an upper bound on solution length) [17]. Felner and Adler further iterated upon on this procedure using *instance dependent pattern databases* [5]. We approach this problem from the opposite position; how can we create and use part of a pattern databases and/or a perimeter over multiple problem instances?

2.3 Perimeter and PDB comparison

Perimeters and pattern databases are similar in many respects. Both perimeters and pattern databases require a predecessor function. This predecessor function enables retrograde search from the goal. Both procedures can also be improved

by using domain-specific properties: perimeters by using a heuristic function between two arbitrary nodes, and pattern databases through symmetry [2], additivity [6], and duality [8, 16].

The two techniques also differ in critical ways. First, pattern databases require a node abstraction mechanism, which perimeters avoid. This freedom allows perimeters to be applied to domains without any known abstraction. On the other hand, where perimeter search requires an available heuristic between any two nodes, pattern databases can generate a heuristic for domains where one is not known. Finally, PDBs cover the entire space, while perimeters only cover part of it (Figure 2). As a result, each node in the perimeter must store a node identifier as well as the cost to the goal. In general, any partial set of the original or abstract space requires extra memory to store the node identifier information. Perimeters fall into this category, as do instance-dependent pattern databases. On the other hand, because pattern databases cover the entire domain space, heuristic values may be indexed by their $nodeID$ (the node need not be stored for each entry (Figure 3)).

2.4 Combining Perimeters with PDBs

Perimeter search works well for correcting pre-existing heuristics [3, 14, 11], while pattern databases prove valuable for creating a heuristic to the goal node [2]. In the remainder of this paper, we propose and investigate a new, general method for combining these two techniques into a single lookup table.

Culberson and Schaeffer use a pattern database as a heuristic simultaneously with a perimeter [1]. Because the pattern database only provides a heuristic to the goal node, perimeter search must use a front-to-goal evaluation technique. If a node is in the perimeter, we know the actual cost to the goal and use that for the heuristic value. A perimeter with cost-bound d has the following property: d equals the minimum cost to the goal of all nodes outside the perimeter. This means that for any node n not inside this perimeter, $h(n) \geq d$ and is corrected accordingly. In fact, as long as d is defined as above, this can be applied to any shaped perimeter.

In a concurrent submission to this SARA symposium, Ariel Felner uses a perimeter to *seed* a pattern database [7]. The pattern database is built using the perimeter as the goal node. This represents an alternative procedure for combining the techniques of perimeter search with pattern databases, but differs significantly from the approach that we present in Section 3.

For every node on the perimeter, Kaindl and Kainz track the difference between the actual cost to the goal and the heuristic value [11]. We call the minimal difference value for all nodes on the perimeter the *heuristic correction factor*. The perimeter is build by expanding the node with the smallest difference, to increase the heuristic correction factor. If the heuristic is consistent, they admissibly add the heuristic correction factor to every node outside the perimeter. A pattern database is a consistent heuristic, so this technique is applicable. However, Felner has shown the following is true for the 15 puzzle using our abstraction method: given an abstract node a with an abstract cost c to the abstract goal, there

exists a node in the original space mapping to a with a cost c from the goal [7]. The same is true for the K-pancake puzzle. Consider using a perimeter built by expanding nodes with the lowest heuristic value [11]. To obtain a correctional factor equal to one, the perimeter must have at least as many entries as the pattern space. Because of the additional memory required by perimeters to store the $nodeID$ (see Figure 3), this method is impractical for our domains.

3 Partial Pattern Databases

Research in pattern databases is beginning to incorporate approaches typically seen in perimeter search. Specifically, subsets of full pattern databases are being stored in the form of *instance-specific* PDBs and caching in hierarchical search [17, 5, 9]. We take this one step further by storing part of a full PDB. Our approach is not instance-specific; it reuses the same database over multiple search instances with a fixed goal.

A *partial pattern database* consists of a set of abstract nodes A and their cost to the goal, where A contains all nodes in S with cost to goal less than d. d is a lower bound on the cost of any abstract node not contained in A. In essence, a partial pattern database is a perimeter in the abstract space (with the interior nodes stored). Any node n in the original space has a heuristic estimate to the goal: if n is in the partial PDB, return the recorded cost; if n is not in the partial PDB, return d. This heuristic is both admissible and consistent.

Building a partial PDB is similar to building a perimeter, only in the abstract space. Retrograde search is performed starting from the abstract goal. The heuristic values are recorded. When a memory limit is reached, the partial PDB building stops and heuristic values are used for the forward search. Depth d is the minimum cost of all abstract nodes outside the partial PDB. Note that all abstract nodes in A with cost equal to d can be removed from the partial PDB to save memory. This will not affect the heuristic.

On one extreme, a partial PDB with no abstraction reverts to exactly a perimeter (with the interior nodes stored). On the other extreme, a partial PDB with a coarse-grained abstraction will cover the entire space, and performs exactly like a full PDB. However, a partial PDB does not store the data as efficiently as a full PDB.

3.1 Memory Requirements

A full PDB encompasses the entire space (Figure 2); every node visited during the forward search has a corresponding heuristic value in the lookup table. The PDB abstraction level is determined by the amount of available memory. Finer-grained abstraction levels are not possible because the memory requirements increase exponentially with finer abstractions.

Partial PDBs, as perimeters, generally do not cover the entire space. During the forward search, if a node is not contained in the partial PDB lookup table, d is returned. Partial PDBs add flexibility over full PDBs by allowing the use

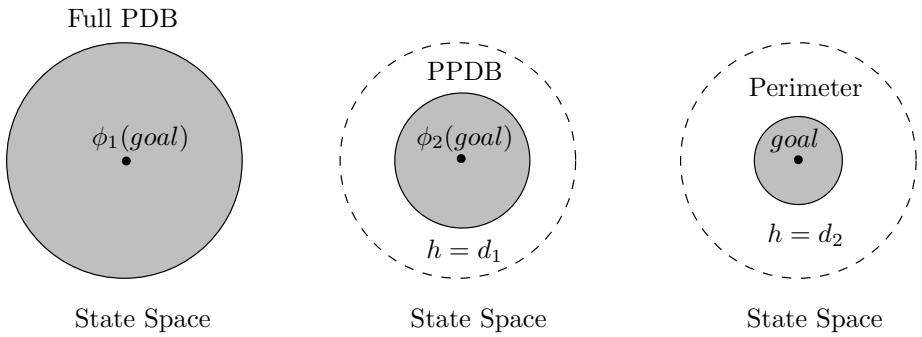


Fig. 2. Coverage of original space by lookup tables (full PDB on left, partial PDB in middle, and perimeter on right)

of any abstraction level. However, there are drawbacks with respect to memory requirements.

Because full pattern databases cover the entire domain space, the heuristic values in the lookup table are indexed by their unique node identifier ($nodeID$). Therefore, a table of the exact size of the abstract space can be used and there is no need to store the $nodeID$ (Figure 3(a)). Memory is only used to store the abstract cost to the goal. On the other hand, perimeters, instance-dependent PDBs [17, 5], and partial PDBs only cover part the space. As a result, each node in the perimeter must store the $nodeID$ in addition to the abstract cost to the goal (Figure 3(b)). This requires extra memory for every table entry.

In our domains, the 15-puzzle and the pancake puzzle, partial pattern database entries require nine times more memory than full pattern database entries. This is a severe limitation on the effective use of partial pattern databases.

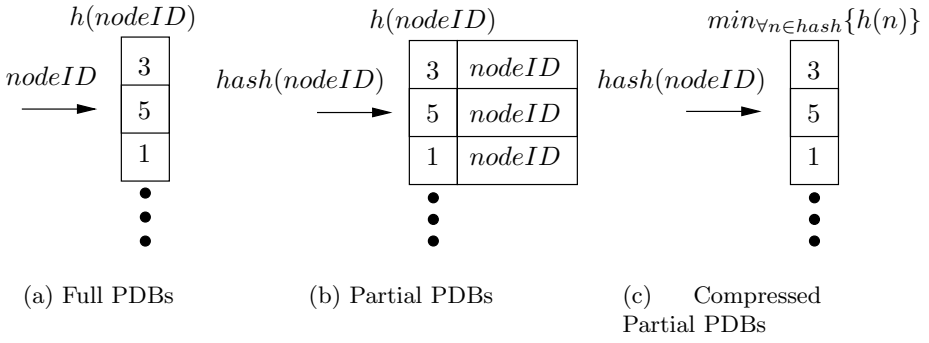


Fig. 3. PDB storage strategies

3.2 Compressed Partial PDBs

For perimeters and partial pattern databases, the added cost of storing a node's identification information is an expensive use of memory. To maintain efficiency, the hash table should have a reasonable fill-factor, but space is inevitably wasted on empty table positions. We therefore present a compressed version of a partial PDB that does not store the $nodeID$ and can be filled to any convenient fill-factor.

Given an abstraction granularity, our hash function maps each abstract node to a location. For all abstract nodes mapped to the same location, we store the minimum heuristic value (to preserve admissibility) (Figure 3(c)). When we query for a node's heuristic value, we return the value stored in the table at that node's hashed location. This heuristic value is guaranteed to be admissible, but it may be inconsistent.

The creation of compressed partial PDBs occurs as a preprocessing step. Therefore, we can take a large amount of time, use machines with more available memory, or use disk-based algorithms. The simplest technique is to use iterative-deepening depth-first search from the abstract goal, filling a value in the table if it is lower than the existing entry. However, unlike with full PDBs, if a node's heuristic value is larger than the entry in the table, it cannot be cut-off without breaking admissibility. Therefore, this *depth-first construction* method must use a complimentary technique to remove transpositions; we use a transposition table [15]. This is the technique used in Section 4.

A second technique is to use breadth-first search. This removes transpositions, but uses a large amount of memory for construction. However, delayed duplicate-detection is an efficient disk-based algorithm that can be used to do this [13]. In the interest of simplicity, we did not attempt this technique.

A third alternative is to build a full PDB at a fine-grained level using a machine with a large amount of memory. Full pattern databases can be computed very efficiently using iterative-deepening depth-first search with only a small amount of excess memory. This is the approach used in Section 5. The main drawback to this approach is that it does not scale well to very fine-grained abstractions. However, our results in Section 4 show that very fine-grained abstractions are not necessary.

One item worth consideration is the hashing function. As the authors found out, a simple modular hashing scheme can introduce regularity in the table. In the worst case, a fine-grained compressed partial PDB can revert to exactly the coarser-grained version. Also, depending on the hash function, the table may not fill to 100%.

4 Experiments on the K-Pancake Puzzle

The abstractions used for the K-pancake puzzle have don't-care tiles as the low-indexed tiles. For example, abstraction-7 for the 12-pancake puzzle refers to the abstraction 'x x x x x 5 6 7 8 9 10 11,' with x as a don't-care. Note that

abstraction-11 is the finest-grained abstraction and is the same as the original space because there are 12 distinct tiles. We use the depth-first construction technique to create the compressed PDBs.

4.1 Performance with constant a number of entries

Throughout this section, we report the average number of nodes generated during the forward search as our metric. Each data point is an average of 100 random starting nodes. Partial pattern databases of various abstraction levels are used for the heuristic. The number of entries in the partial PDB remains constant (specified in the results tables), while the level of abstraction varies. IDA* with BPMX is used for the forward search. Recall that BPMX only has an effect if the heuristic is inconsistent; therefore, only the compressed partial pattern database will be affected by using BPMX. Also, note that this is only a comparison between partial pattern databases; full pattern databases make much more efficient use of memory. We will cross-compare techniques in Section 4.2.

Table 1 shows the average number of nodes generated on the 8, 10, 12, and 13-pancake puzzles. Each puzzle fixes the number of entries in the partial PDB to exactly the number of entries of a full PDB with $\lfloor K/2 \rfloor$ unique tiles. For instance, the 12-pancake puzzle partial PDBs each have 665,280 entries (six unique tiles). The top data point of each column generates the same number of nodes as a full pattern database, while the bottom data point generates the same number of nodes as a perimeter.

Number of Unique Tiles	8-Pancake 1,680 entries (cost-bound d)	10-Pancake 30,240 entries (cost-bound d)	12-Pancake 665,280 entries (cost-bound d)	13-Pancake 1,235,520 entries (cost-bound d)
4	2,065 (7)			
5	682 (5)	48,408 (9)		
6	403 (5)	14,268 (6)	1,316,273 (11)	25,833,998 (12)
7	**335 (5)**	8,251 (6)	178,464 (8)	**3,106,345 (8)**
8		7,370 (6)	183,172 (7)	4,097,683 (7)
9		**7,242 (6)**	167,584 (7)	3,851,260 (7)
10			165,390 (7)	3,820,667 (7)
11			**164,951 (7)**	3,816,931 (7)
12				3,819,909 (7)

Table 1. Average number of nodes generated using a single Partial PDB on K-Pancake puzzle.

We see that with the same number of entries, but using a finer granularity partial PDB, the average number of nodes generated reduces. Note however, that in the 12-pancake puzzle, abstraction-7 produced fewer nodes than abstraction-8. Search performance is very sensitive to the cost-bound of the partial pattern

database. The increase in nodes generated from abstraction-7 to abstraction-8 occurs because abstraction-7 has a larger cost-bound d than abstraction-8.

Consider two partial pattern databases with the same cost-bound d, but one being based on a coarser granularity than the other. The finer-grained database will *dominate* the coarser-grained one because for every node n in the space, $h_{fine}(n) \geq h_{coarse}(n)$. Abstraction-8, 9, 10, and 11 on the 12-pancake puzzle demonstrate this principle.

We will continue to see this trade-off between abstraction granularity and database cost-bound throughout the results. We can think of this intuitively as follows: making a partial PDB finer-grained improves the heuristic value of nodes inside the database at the cost of nodes outside the database (if d gets smaller). At some point the heuristic outside the partial PDB becomes so inaccurate that the performance is dominated by these nodes. Refining the abstraction further from this point generally produces worse performance. Analogous results are documented in [10], where increasing small heuristic values improves performance, but only up to a point.

Number of Unique Tiles	8-Pancake 1,680 entries (cost-bound d)	10-Pancake 30,240 entries (cost-bound d)	12-Pancake 665,280 entries (cost-bound d)	13-Pancake 1,235,520 entries (cost-bound d)
4	2,065 (7)			
5	**1,024 (6)**	48,414 (8)		
6	1,227 (6)	**18,026 (8)**	1,316,284 (10)	25,834,132 (10)
7	1,564 (6)	22,117 (7)	**358,585 (9)**	**6,481,829 (9)**
8		29,867 (7)	379,655 (9)	6,599,913 (9)
9		39,080 (7)	520,648 (8)	10,142,002 (9)
10			677,805 (8)	15,214,336 (8)
11			841,576 (8)	22,068,332 (8)
12				27,498,382 (8)

Table 2. Average number of nodes generated using a single compressed partial PDB filled to 70% on K-Pancake puzzle.

Table 2 shows the average number of nodes generated over 100 search instances using compressed partial PDBs. The compressed database is filled to 70% full. We examine the 8, 10, and 12-pancake puzzles.

The top entry of each column matches closely with the performance of a full pattern database (shown at the top of the corresponding columns in Table 1). At this abstraction level, each entry corresponds to one abstract node; if the compressed partial PDB were filled to 100% and had no hash collisions, then it would be identical to a full PDB. However, these conditions do not hold in general, so the top entries in each table do not match exactly.

Let us examine another two corresponding entries in each table, 12-Pancake at abstraction-7. For the partial pattern database (Table 1), there is an average

of 178,464 nodes generated. For the compressed partial PDB (Table 2), there are 358,585 generated nodes on average. Keep in mind that both tables have the same number of entries, but the entries themselves are different.

The compressed partial PDB generates more nodes for three reasons. First, the table is only filled to 70% full, but this has a small effect because the unreached entries are filled with d, limiting the heuristic error. Second, the heuristic values in the perimeter are degraded by taking the minimal value of all nodes hashed to the same location. Third and most importantly, the heuristic correction of d is not applied to all nodes outside the perimeter. Because we store the minimum heuristic value, nodes outside the perimeter overlap with nodes inside the perimeter, reducing the effect of the heuristic correction factor.

However, because of the construction method, the partial PDB does not necessarily dominate the compressed partial PDB. The compressed partial PDB is filled until 70% full. Overlapping nodes cause the table to fill more slowly than the partial PDB. Thus the final cost-bound d may be greater in the compressed partial PDB than the partial PDB. This is seen in our example with the 12-Pancake puzzle at abstraction-7: the partial PDB is built to $d = 8$ while the compressed partial PDB is built to $d = 9$.

As the granularity becomes finer, we quickly reach the point of diminishing returns. In all examples, this occurs after adding one more unique tile to the original PDB abstraction. For the 8, 10, 12, and 13-pancakes, the optimal granularity is 5, 6, 7, and 8 unique tiles respectively. Further refining of the abstraction only causes the number of generated nodes to increase. This is caused by the poor, high-valued heuristics.

The *improvement factor* is the improvement over the original abstraction (top entry in Table 1). This factor increases with puzzle size. The 8, 10, 12, and 13-pancake puzzles' best performance factors are 50%, 63%, 73%, and 75%. This indicates that savings may scale favorably to larger problems.

4.2 Performance with constant memory

As stated, partial pattern databases need extra memory to store the $nodeID$. In the case of the 10-pancake puzzle, the amount of memory used to store the $nodeID$ is about eight times larger than the heuristic value that is stored. To directly compare partial PDBs with full PDBs and compressed partial PDBs, we need to keep memory constant. So we limit the number of entries in the partial PDB appropriately.

Each full PDB entry consists of one byte, as does each compressed partial PDB entry. Each partial PDB entry consists of 9 bytes (1 for the heuristic value and 8 for the node id) plus extra room in the hash table for empty entries (fill factor). We therefore calculate an approximate number of entries for the partial PDB that fits into the designated number of bytes using the formula: $entries_{partialPDB} = bytes/9 * 0.7$.

Table 3 directly compares the performance of the three heuristic techniques (full PDBs, partial PDBs, and compressed partial PDBs) on the 10-pancake puzzle while keeping memory constant. Each data entry is the average number of

10-Pancake Puzzle			
Memory Limit	Normal PDB (unique tiles)	Best Partial PDB (unique tiles)	Best Compressed Partial PDB (unique tiles)
3,628,800 bytes	**40 (9)**	2,003 (7)	**40 (9)**
1,814,400 bytes	200 (8)	4,628 (7)	**70 (9)**
604,800 bytes	1,216 (7)	16,147 (6)	**417 (8)**
151,200 bytes	7,756 (6)	78,073 (5)	**2,695 (7)**
30,240 bytes	48,408 (5)	511,794 (5)	**18,026 (6)**
5,040 bytes	337,021 (4)	3,299,716 (4)	**135,352 (5)**

Table 3. Average number of nodes generated using a single PDB of the best abstraction granularity. The tests are performed on the 10-Pancake puzzle while keeping memory constant.

generated nodes over 100 random instances. For the partial PDB and compressed partial PDB, we tested from one to nine unique tiles; this table shows only the best result. The associated number in parentheses depicts the number of unique tiles used to generate the data point. For each memory limit, we show the best-performing database in bold.

Partial PDBs by themselves are not an efficient use of memory; as in all tested cases, partial PDBs generate at least an order of magnitude more nodes than full PDBs. However, the compressed partial PDBs are an efficient use of memory. The top row covers the entire space at the finest granularity; this is a perfect heuristic. In this case the full PDB has slightly better performance because the compressed partial PDB is only filled to 70% (this is not apparent in the table because of averaging). The improvement factor for every row except the first is between 60% and 65%, indicating similar performance gains when the memory limit is smaller than the size of the abstract space.

5 Experiments on the 15-Puzzle

The abstractions used for the 15-puzzle are as follows: abstraction-8 is the *fringe* [2], 'b x x 3 x x x 7 x x x 11 12 13 14 15'; and abstraction-9 adds one more unique tile, 'b x x 3 x x x 7 x x 10 11 12 13 14 15'. The heuristic used is the maximum of Manhattan Distance and the PDB or compressed partial PDB lookup. No symmetries or other search enhancements are used.

Each test is run over all 100 Korf problem instances [12]. The databases compared are the full pattern database with abstraction-8 (PDB_8) and the compressed partial pattern database using abstraction-9 ($PPDB_9$). $PPDB_9$ is created from the full PDB using abstraction-9. We use IDA* with bidirectional pathmax (BPMX) to take advantage of the inconsistency in the compressed partial PDBs.

The columns of Table 4 are as follows:

– The *PDB* is the type of pattern database used: either the full pattern database PDB_8 or the compressed partial pattern database $PPDB_9$.

PDB	Memory	BPMX	Fill	d	small	medium	large	total
PDB_8	518,918,400	DC	100	64	283,309	6,929,803	80,291,808	1,067,439,170
$PPDB_9$	518,918,400	Yes	50	38	792,425	14,465,508	123,546,247	1,840,334,929
$PPDB_9$	518,918,400	Yes	60	39	651,152	12,181,920	101,463,007	1,525,898,811
$PPDB_9$	518,918,400	Yes	70	40	554,045	10,529,666	86,096,055	1,304,112,453
$PPDB_9$	518,918,400	Yes	80	41	487,551	9,361,520	75,459,248	1,149,450,912
$PPDB_9$	518,918,400	Yes	90	43	409,084	7,943,644	62,933,272	965,285,000
$PPDB_9$	518,918,400	Yes	98	66	330,510	6,473,794	50,611,367	780,456,393
$PPDB_9$	518,918,400	No	98	66	539,065	9,917,842	82,560,198	1,242,278,013

Table 4. Nodes generated on the 15-puzzle using a single PDB technique and Manhattan Distance while keeping memory constant.

- *Memory* is the number of bytes in the database: in this case held constant at 518,918,400 bytes.
- *BPMX* tells whether bidirectional pathmax is used: *Yes* it is used, *No* it is not used, and *DC* (don't care) means that BPMX has no effect.
- *Fill* shows the percentage of memory that is expanded when creating the pattern database. Because of the hashing scheme, however, even when filled completely, 2% of $PPDB_9$ remains unexpanded.
- For PDB_8, d is the largest value in the database. For $PPDB_9$, d is the cost bound (as defined in Section 3).
- The *small* problems are the problems that result in searches with less than 1,000,000 generated nodes when solving with PDB_8. *Medium* problems are between 1,000,000 and 31,999,999 generated nodes. The *hard* problems are greater than or equal to 32,000,000 generated nodes. There are 43 small problems, 48 medium problems, and 9 hard problems. We report the average number of nodes expanded in each of the three problem sets.
- *total* is the sum of all generated nodes over the 100 Korf problem instances.

At less than 90% full, using BPMX, $PPDB_9$ generates more total nodes than PDB_8. $PPDB_9$ generates slightly fewer nodes when at 90% full, and at 98% full the total number of generated nodes is decreased by 27%. However, this result can be slightly misleading, since the total is dominated by the largest searches (which build over three orders of magnitude larger search trees). Thus, we also examine the problems grouped by difficulty. With and without BPMX, the small problems have worse performance using $PPDB_9$ than PDB_8. However, the large problems have improved performance when using BPMX and filled to 80% or more. When $PPDB_9$ is 98% full, the small, medium, and large problems have -95%, 6%, and 36% improvement respectively, in average number of nodes generated. Not only does the total number of generated nodes decrease by 27%, but the hard instances are improved the most.

The last row shows the importance of using BPMX in the 15-puzzle to improve performance. Using BPMX leads to a 37% reduction in the total number of nodes generated when $PPDB_9$ is 98% full. This pushes the performance ahead

of PDB_8. Small, medium, and large instances benefit equally from BPMX, improving the number of generated nodes by 39%, 35%, and 39% respectively. This indicates that the influence of BPMX may not depend on instance difficulty.

6 Conclusions

This paper presents partial pattern databases, a general approach that merges the ideas of front-to-goal perimeter search and full pattern databases. Our approach decouples the abstraction granularity from the database size, freeing the programmer to use the best abstraction for a given domain and amount of memory. Partial PDBs are applicable to domains with a predecessor function and a space abstraction technique and can be re-used over multiple problem instances with the same goal.

Two versions are presented: the original partial pattern databases, which store the heuristic value and the $nodeID$; and the more memory-efficient compressed partial PDBs, which store only one heuristic value. Two complementary puzzle domains are tested: the K-pancake puzzle, which has a large branching factor of $K-2$, and the 15-puzzle, which has an average branching factor of 2.1.

Compressed partial pattern databases are shown to be most effective on the K-pancake puzzle; the number of nodes generated on the 13-pancake puzzle is reduced by three-fold. Uncompressed partial pattern databases are not shown to be effective for this domain because of the memory inefficiency.

On the 15-puzzle, compressed partial PDBs in combination with the Manhattan Distance heuristic are slightly less successful. This is because MD often corrects a PDB's low heuristic values, which is one of the primary benefits of using partial PDBs. However, in combination with BPMX, and filled to 98%, we are able to reduce total number of nodes generated by 27%. Interestingly, hard problem instances are better improved than small instances.

This paper presents, implements, and tests partial PDBs in general terms. This technique can be further incorporated with other general methods. For example, the maximum can be taken over multiple heuristic lookup tables, whether they be PDBs, partial PDBs, or compressed partial PDBs [10]. As well, domain-specific adaptations and improvements can be integrated into this framework. Two glaring examples are additivity in the 15-puzzle and duality in the pancake puzzle. On the 15-puzzle, partial PDBs can always be made into additive versions by ignoring don't-care tile movements. Compressed partial PDBs can effectively compress larger full additive pattern databases into memory-efficient versions. On the pancake puzzle, we can use any pattern database technique to get a heuristic to the goal. By using the *general duality principal* [8], we can get get an admissible heuristic between any two nodes (map the operator sequence Π between two nodes onto the goal to get node n^d, and look up $h(n^d)$ from the PDB). This would allow for front-to-front heuristic improvement using a partial pattern database, effectively coming full-circle back to the original perimeter search technique.

7 Acknowledgments

We would like to express our thanks and appreciation to the reviewers for their comments and corrections, Cameron Bruce Fraser and David Thue for proof-reading, and Ariel Felner for his thought-provoking discussions. This work was supported by NSERC and iCORE.

References

1. Joseph C. Culberson and Jonathan Schaeffer. Efficiently searching the 15-puzzle. Technical Report TR 94-08, Department of Computing Science, University of Alberta, 1994.
2. Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Canadian Conference on AI*, pages 402–416, 1996.
3. John F. Dillenburg and Peter C. Nelson. Perimeter search. *Artificial Intelligence*, 65(1):165–178, 1994.
4. Stefan Edelkamp. Planning with pattern databases. In *ECP: European Conference on Planning*, pages 13–34, Toledo, 2001.
5. Ariel Felner and Amir Adler. Solving the 24 puzzle with instance dependent pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 248–260, 2005.
6. Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *JAIR: Journal of Artificial Intelligence Research*, 22:279–318, 2004.
7. Ariel Felner and Nir Ofek. Combining perimeter search and pattern database abstractions. In *Symposium on Abstraction Reformulation and Approximation (SARA)*, 2007.
8. Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert Holte. Dual lookups in pattern databases. In *IJCAI*, pages 103–108, 2005.
9. Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical heuristic search revisited. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 121–133, 2005.
10. Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple pattern databases. In *ICAPS*, pages 122–131, 2004.
11. Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.
12. Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artifificial Intelligence*, 27(1):97–109, 1985.
13. Richard E. Korf. Delayed duplicate detection: Extended abstract. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1539–1541. Morgan Kaufmann, 2003.
14. Giovanni Manzini. BIDA: An improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.
15. Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intellgence*, 16(7):701–710, 1994.
16. Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. In *AAAI*, pages 1076–1081, 2006.
17. Rong Zhou and Eric A. Hansen. Space-efficient memory-based heuristics. In *AAAI*, pages 677–682, 2004.
18. Rong Zhou and Eric A. Hansen. External-memory pattern databases using structured duplicate detection. In *AAAI*, pages 1398–1405, 2005.