Solving the 24 Puzzle with Instance Dependent Pattern Databases

Ariel Felner[1] and Amir Adler[2]

[1] Dpet. of Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva, 84104, Israel
EMAIL: felner@bgu.ac.il
[2] Dept. of Computer Science, Technion, Haifa, 32000, Israel
EMAIL: adlera@cs.technion.ac.il

Abstract. A *pattern database* (PDB) is a heuristic function in a form of a lookup table which stores the cost of optimal solutions for instances of subproblems. These subproblems are generated by abstracting the entire search space into a smaller space called the pattern space. Traditionally, the entire pattern space is generated and each distinct pattern has an entry in the pattern database. Recently, [10] described a method for reducing pattern database memory requirements by storing only pattern database values for a specific instant of start and goal state thus enabling larger PDBs to be used and achieving speedup in the search. We enhance their method by dynamically growing the pattern database until memory is full, thereby allowing using any size of memory. We also show that memory could be saved by storing hierarchy of PDBs. Experimental results on the large 24 sliding tile puzzle show improvements of up to a factor of 40 over previous benchmark results [8].

1 Introduction

Heuristic search algorithms such as A* and IDA* find optimal solutions to state-space search problems. They visit states in a best-first manner according to the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the actual distance from the initial state to state n and $h(n)$ is a heuristic function estimating the cost from n to a goal state. If $h(s)$ is "admissible" (i.e., is always a lower bound) then these algorithms are guaranteed to find optimal paths.

The *domain* of a search space is the set of constants used in representing states. A *subproblem* is an *abstraction* of the original problem defined by only considering some of these constants and mapping the rest to a "don't care" symbol. A *pattern* is a state of the subproblem. The abstracted *pattern space* for a given subproblem is a state space containing all the different patterns connected to one another using the same operators that connect states in the original problem. A *pattern database* (PDB) stores the distance of each pattern to the goal pattern. These distances are used as admissible heuristics for states of the original problem by mapping (abstracting) each state to the relevant pattern in the pattern database.

Typically, a pattern database is built in a preprocessing phase by searching backwards, breadth-first, from the goal pattern until the whole abstracted pattern space is

spanned. Given a state S in the original space, an admissible heuristic value for S, $h(S)$, is computed using a pattern database in two steps. First, S is mapped to a pattern S' by ignoring details in the state description that are not preserved in the subproblem. Then, this pattern is looked up in the PDB and the corresponding distance is returned as the value for $h(S)$. The value stored in the PDB for S' is a lower bound (and thus serves as an admissible heuristic) on the distance of S to the goal state in the original space since the pattern space is an *abstraction* of the original space. PDBs have proven very useful in optimally solving combinatorial puzzles and other problems [1, 7, 8, 3, 6, 2].

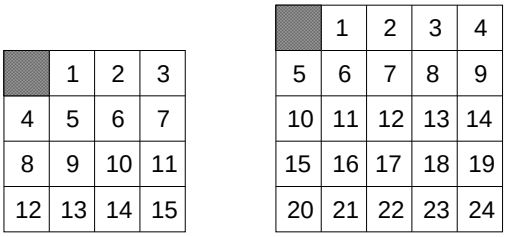

Fig. 1. The 15 and 24 Puzzles in their Goal States

The 15 and 24 tile puzzles are common search domains. They consist of 15 (24) numbered tiles in a 4×4 (5×5) square frame, with one empty position - the *blank*. A legal move swaps the *blank* with an adjacent tile. The number of states in these domains is around 10^{13} and 10^{24} respectively. Figure 1 shows these puzzles in their goal configurations.

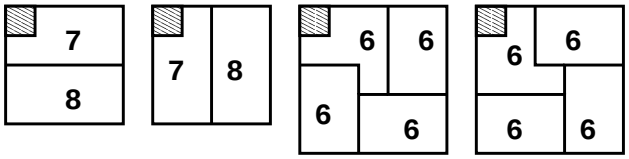

Fig. 2. Partitionings and reflections of the tile puzzles

The best existing optimal solver for the tile puzzles uses *disjoint PDBs* [8]. The tiles are partitioned into disjoint sets (subproblems) and a PDB is built for each set. Each PDB stores the cost of moving *only* the tiles in the given set from any given arrangement to their goal positions and thus values from different disjoint PDBs can be *added* and are still admissible. An $x - y - z$ partitioning is a partition of the tiles into disjoint sets with cardinalities of x, y and z. [8] used a 7-8 partitioning for the 15 puzzle and a 6-6-6-6 partitioning for the 24 puzzle. These partitionings were reflected about the main diagonal (as shown in figure 2) and the maximum between the regular and the reflected PDB was taken as the heuristic.

The speed of search is inversely related to the size of the PDB used, i.e., the number of patterns it contains[5]. Larger PDBs take longer to compute but the main problem is the memory requirements. With a given size of memory only PDBs of up to a fixed size can be stored. Ordinary PDBs are built such that they are randomly accessed. Thus, storing larger PDB on the disk is impractical and would significantly increase the access time for a random PDB entry unless a sophisticated disk storage mechanism is used. For example a mechanism built on the idea of [11] can be used for storing PDBs on disk. Note, however, that even if the PDBs are stored on disk, disk space is also limited.

A possible solution for this was suggested by [4]. They showed that instead of having a unique PDB entry for each pattern, several adjacent patterns can be mapped to only one entry. In order to preserve admissibility, the compressed entry stores the minimum value among all these entries. They showed that since values in PDBs are locally correlated most of the data is preserved. Thus, we can build large PDBs and compress them into smaller sizes. A significant speedup was achieved using this method for the 15 puzzle and the 4-peg Towers of Hanoi problems.

There are, however, a number limitations to this technique. First, the entire pattern space needs to be generated. Second, only a limited degree of compressing turned out to be effective. For the tile puzzles, it was only beneficial to compress pairs of patterns achieving a memory fold of a factor of two. The largest PDBs that could be built using this technique for the 24 puzzle with one gigabyte of main memory was a $5-5-7-7$ partitioning where the 7-tile PDBs were compressed by a factor of two. This did not gain a speedup over the $6-6-6-6$ partitioning of figure 2 which is probably the best 4-way partitioning of the 24 puzzle.

The motivation for this paper is to use larger PDBs for the 24 puzzle. We want at least an $8-8-8$ partitioning of this domain. A pattern space for 8 tiles has $25\times 24\ldots\times 18 = 4.36\times 10^{10}$ different patterns. Storing three different complete 8-tile PDBs would need 130 gigabytes of memory!!! Using the compressing idea of [4] would not help much and alternative idea should be used.

Another way of achieving reduction in memory requirements is to build a PDB for a specific instance of a start and goal states. Some recent works used this idea for solving the multiple sequence alignment problem, e.g., [9] where the PDB was stored as an Octree. A general formal way for doing this was developed by [10]. They showed that for solving a specific problem instance only a small part of the pattern space needs to be generated. In this paper, we call this idea *Instant dependent pattern databases* (IDPDB). We suggest a number of general enhancements and simplifications to this method and apply them to the 24 puzzle. Experimental results show a reduction of up to a factor of 40 in the number of generated nodes for random instances of this puzzle.

2 Instant dependent pattern databases

We first want to distinguish between the *original search space* where the actual search is performed from the *pattern space* which is a projection of the original search space according to the specification of the patterns (see figure 3). Solving a problem involves two phases. The first phase builds the PDB by performing a breadth-first search back-

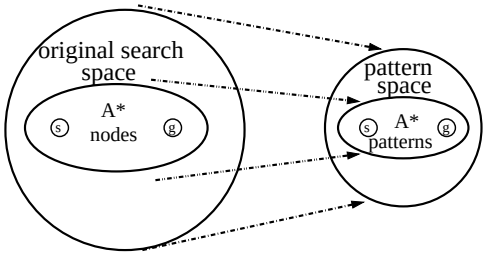

Fig. 3. The projection/abstraction into the pattern space.

wards from the goal pattern until the entire pattern space is spanned. The second phase performs the actual search in the original search space.

Traditionally, a PDB has a unique entry for each possible pattern. [10] observed that for a given instance of start and goal states only nodes generated by A* (or IDA*) require a projected pattern entry in the PDB since only these nodes are queried for a heuristic value during the search. In this paper we call these nodes the *A* nodes* and their projections the *A* patterns* (see figure 3). Ideally, we would like to identify and only generate the exact set of the A* patterns but this is impossible. They defined a *focused memory-based heuristic* as a memory based-heuristic (PDB) that is computed only for patterns that are projections of states in the original search space that could be explored by A* in solving the original search problem.

For building the PDB they also search backwards from the goal pattern but are focused on the specific start and goal states. Instead of the usual breadth-first search which searches in all possible directions they activate A* from the goal pattern to the start pattern. In this paper, we call it the *secondary A** in order to distinguish this search from the *primary search* in the original problem which could be performed by any admissible search algorithm (e.g., IDA*). For each pattern expanded by the secondary A*, its g-cost represents the cost of a shortest path from this pattern to the goal pattern in the pattern space and can serve as an admissible h-cost for the original search problem.

After the start pattern is reached by the secondary A* search only a small number of patterns were generated and there is no guarantee that the entire set of the A* patterns was reached. They noted that we can continue to expand nodes after the secondary A* finds an optimal path from the goal pattern to the start pattern to determine optimal g-costs (and thus admissible heuristics) for additional states. We call this the *extended secondary A** phase. We would like to halt the extended secondary A* phase when all the A* patterns are reached but this is a difficult task. They provide a method for identifying a special set of patterns which is a superset of the A* patterns set. We call this set the *ZH set* (after Zhou and Hansen). They halt the extended secondary A* phase after the complete ZH set is generated and are guaranteed that the entire set of the A* patterns is generated.

The definition of the ZH set is as follows. Let U be an upper bound on the cost of the optimal solution to the original problem. The ZH set includes all patterns p_i which have $f(p_i) < U$ in the secondary A* search. It is obvious that all the A* patterns have f-value in the secondary search smaller than U and thus are all included in the ZH

set. They also provide a formula for computing and generating the ZH set for a set of disjoint additive PDBs. Let U again be an upper bound on the solution. Let $L = \sum_j(h_j(S))$ be the additive heuristic of the initial state S. Let $\Delta = U - L$. They prove that a disjoint PDB, PDB_j only needs to be calculated for projected patterns p_i having $f(p_i) < h_j(S) + \Delta$. See [10] for more details and proofs.

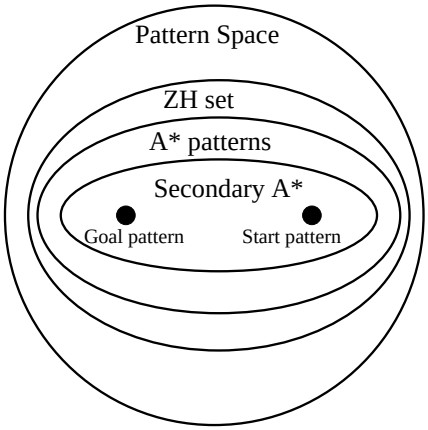

Fig. 4. The different layers of the pattern space.

Figure 4 shows the relations between different sets of patterns. The innermost set includes the patterns generated during the first stage of secondary A* (until the optimal path from the goal pattern to the start pattern is found). The next set includes the A*-patterns, i.e., the patterns queried during the search in the original problem. The next set includes the ZH set which [10] stored in their PDB. The outmost set is the complete pattern space.

They recognized that continuing the extended-A* until the complete ZH set is generated is not always possible due to time/memory limitations. Thus, they introduced their γ factor where $0 < \gamma \leq 1$. They stopped the extended-A* when its f-cost exceeds $\gamma \times U$. With $\gamma < 1$ generating all the A* patterns is not guaranteed. Therefore, when a state on the original space is reached whose projected pattern is not in the PDB due to the γ cutoff, they suggest using a simple quickly computed admissible heuristic instead. They implemented their idea with different values for γ on the multiple-sequence alignment and obtained impressive results.

Note that ordinary PDBs are usually stored in a multi-dimensional array with an entry for each possible pattern. For IDPDB, we need a more sophisticated data structure e.g., a hash table, as only a subset of the patterns is stored.

2.1 Weaknesses of the ZH method.

There are a number of weaknesses in the ZH set approach.

1) Their method needs a fixed amount of memory. Once γ is chosen all the nodes with $f \leq \gamma \times U$ are stored. This is problematic as it is difficult to determine the exact value for γ so that the available memory would be fully used and not be exhausted. Identifying the ZH set and then simplifying it by γ seems not natural and ad hoc.

2) An upper bound, U, for an optimal solution to the original search problem cannot always be found. Furthermore, we need a strict upper bound so as to reduce the ZH set as much as possible.

3) [10] tried out their method only on multiple sequence alignment. The search space of this domain (and also the projected pattern space) has the property that the number of nodes in a given depth d is polynomial in d. This is because the problem is formalized as an n-dimensional lattice with l^n nodes where n is the number of sequence to be aligned and l is the length of the sequences. Even with relatively mediocre U bounds, their ZH set might be significantly smaller than the entire pattern space. This is not true in domains such as the tile puzzles where the number of nodes at depth d is exponential in d. Given any U the ZH set might include the entire domain.

To support these claims experimentally we applied the formula they provide for creating the ZH set for disjoint PDBs on the 15 puzzle. This formula uses an upper bound on the optimal solution. We used the best upper bound possible - the exact optimal solution. We calculated the ZH set with this strict upper bound for a 5-5-5 and a 6-6-3 partitionings of the 15 puzzle on the same 1000 random instances from [8]. The entire 5-5-5 PDB includes 1,572,480 entries, half of them were queried during the search. The average ZH set over the 1000 instances has 1,227,134 pattern - 78% of the entire pattern space. For many difficult instances of this domain, the ZH set actually included the entire pattern space. On those instances the ZH method is useless. Note again that this is when we used a strict upper bound of the actual optimal solution length. For more realistic larger upper bounds the ZH will be even larger. Similar results were obtained for a 6-6-3 partitioning.

3 Dynamically growing the PDBs

We suggest the following enhancement to Zhou and Hansen's idea. Our enhancement is at least as strong as their method but is simpler to implement and easier to understand. In addition, our idea can fit any size of available memory.

The main point of our idea is to dynamically grow the PDB until main memory is exhausted. Our idea is much more flexible than the method of [10] as it can work with any size of memory and we do not need to decide when to halt the secondary A* extension in advance. Furthermore, we do not need to calculate any upper bounds nor have to build the ZH set. In the preprocessing phase, we continue generating patterns in the extended secondary A* until memory is exhausted. We then start the primary search phase and for each pattern not in the PDB, we use a simple quickly computed admissible heuristic instead.

The following enhancement can better utilize main memory after it was exhausted. There are two data structures in memory. The first is the PDB which is identical to the closed list of the extended secondary A*. The second is the open-list of the extended secondary A*. However, at this point we can remove the open-list from main memory

thus freeing a large amount of memory for other purposes such as other PDBs. In fact, if x is the f-value of the best node in the open list and is also the value of the last node expanded, then all nodes in the open-list with values of x can be added to the PDB before freeing the memory. This is actually expanding these nodes without actually generating their children.

Another way of saving memory is to use IDA* for the secondary search. Here, each new pattern generated is matched against the PDB and if it is missing a new PDB entry is created. However, in the pattern space of eight tiles presented below there are many small cycles since all the other tiles can be treated as blanks. This causes IDA* to be ineffective in this specific pattern space because it cannot prune duplicate nodes due to its depth-first behavior.

3.1 On Demand pattern databases

A version of the above idea is called *on-demand pattern database*. Here, we add patterns to the PDB only when they are required during the search. This prevents us from generating large PDBs with patterns that will not be queried.

First, we run the secondary A* from the goal pattern to the start pattern until the start pattern is chosen for expansion. Each pattern expanded by this search is inserted into the PDB. At this point, the preprocessing phase ends and the primary search can begin because the start pattern is already in the PDB. We continue the primary search as long as projected patterns of new nodes are in the PDB (i.e., were expanded by the secondary A* search). When we reach a pattern p not in the PDB and still have free memory, we continue to extend the secondary A* phase until this pattern p is reached and we can return to the primary A* phase.

When memory is exhausted the PDB has reached its final size and the secondary A* is terminated. From this point, each time a heuristic is needed and the relevant pattern is not in the PDB, we consult the quickly computed heuristic.

4 Implementation on the 24 puzzle

While the above idea is a general one we made some domain dependent enhancements and took special steps to best fit the IDPDB idea to the 24 puzzle.

Generating a PDB consumes time. However, the time overhead of preprocessing the PDBs is traditionally omitted as it is claimed that it can be amortized over the solving of many problem instances. For example, it takes a couple of hours to generate the 7-8 disjoint PDB which was used to solve the 15 puzzle [8]. Yet, the authors ignored this time and only reported the time of the actual search which is a fraction of a second.

We cannot simply omit the time overhead of generating IDPDBs as a new PDB has to be built for each new instance. Therefore, it is irrelevant to apply this idea to small domains such as the 15 puzzle where the running time of the actual search is much smaller than the time overhead of generating the PDB. We cannot see how this method will improve previous running times for such domains. The 24 puzzle is a different story since it is 10^{11} times larger. Generating the 6-6-6-6 PDB also takes a number of hours. However, a number of weeks were required to solve many of the instances of [8].

Here, the time overhead of generating the PDB can also be omitted when compared to the overall time needed to solve the entire problem.

The above general method is for generating one PDB. When we use disjoint databases, such as the 8-8-8 partitionings for the 24 puzzle (see below), values from the different PDBs are added and therefore three values for each state of the original search space are required. Thus, the on-demand version of IDPDB activates three secondary A* searches in parallel, one for each PDB[3]. Note, that since each move in the tile puzzle domain moves only one tile then at each step we only need to consult the one PDB that includes the tile that has just been moved. Values from the other PDBs can be inherited from the parent and remain identical.

4.1 On Demand versus preprocessing

The weakness of the on-demand approach for the 24 puzzle is that three open-lists are maintained at all times but this is wasteful since the open-list can be deleted after memory is exhausted. In the special case of the tile puzzles an open list might have 10 times more nodes than the closed list.

A better way to utilize memory for this domain is to perform the complete secondary A* in the preprocessing phase until memory is exhausted. At this point the closed list which includes all the patterns with valid heuristics is stored in a file on the disk and the entire memory is released. This mechanism is repeated for each PDB until a relevant file with heuristic values is stored on the disk. Memory is better utilized as only one open-list is maintained in memory at any point of time. Furthermore, during the course of the primary search there are no open lists of the secondary searches in main memory. Now, we can load values from the disk files into memory and have a PDB for each of them.

In both the on-demand and the preprocessing variations when an entry was missing from the PDB we took the Manhattan distance (MD) as an alternative simple heuristic for the tiles of the missing entry. In addition, for most variations reported below we also stored the benchmark 6-6-6-6 PDBs (which needed 244 megabytes). We then compared the heuristic obtained from the 8-8-8 PDB to the 6-6-6-6 heuristic and took the maximum between them.

4.2 Improvement 1: Internally partition the PDB

Note that each of the 8-tile sets of figure 5 can be internally partitioned into a 6-2 partitioning where the 6-tile partition is one of the 6-6-6-6 partitions of figure 2. For example, the 6-2 partitioning is shaded in gray for partition a of figure 5. Instead of taking the MD for the eight tiles of a missing entry, we can use the 6-2 partition of these tiles. For those eight tiles we added the value of the corresponding 6-tile pattern from the 6-6-6-6 PDB to a value of the 2 tiles from a new 2-tile PDB which was also generated.

[3] Furthermore, this is true for using other combinations of multiple PDBs such as taking the maximum over different PDB values

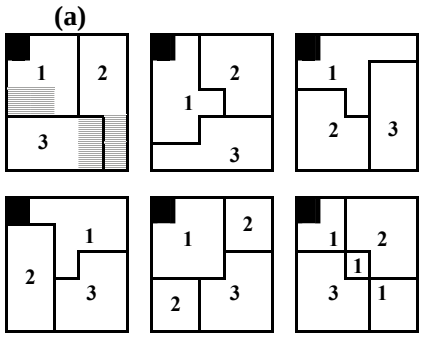

Fig. 5. Different Partitioning to 8-8-8

4.3 Improvement 2: Hierarchical PDBs

Note that once there is a simple heuristic in hand, the new PDB heuristic only requires storing entries for patterns which their PDB values are larger than the simple heuristic. This suggests an hierarchy of PDBs. First, you store a small weak PDB. Then, for the stronger PDB you store only those entries having values larger than the weaker PDB. We used this idea as follows. For any 8-tile PDB we only need values which are larger than the 2-6 partitioning described above. Values of the 8-tile PDB which are not larger can be omitted and retrieved from the 6-2 PDBs. This was very effective as only 18% of the values of an 8-tile PDB were larger than the corresponding 6-2 PDB and had to be stored. The overhead for this was the need to generate and store the relevant 6-tile and 2-tile PDBs but we stored the 6-6-6-6 PDBs anyway as described above and the overhead of generating and storing a number of 2-tile PDBs is very low.

Here we only stored two levels using this hierarchical approach. Future work can take this further by building an hierarchy of PDB heuristics where each PDB is built on top of the previous one in the hierarchy. Note that the basics of this approach were used in [8] for the 15 puzzle where the weaker heuristic was MD and only additions above MD were stored in the PDB. Thus, values for patterns equal to MD were stored as 0. Here we further improve on this approach and omit values of 0.

4.4 Improvement 3: Multiple partitioning

The bottleneck for the IDPDB method is the memory requirements of the secondary A* search. This phase terminates when memory is exhausted. The memory requirements for the primary search phase is much smaller especially after applying all the improvements above. It is well known that PDBs are better utilize by having multiple PDBs and taking the maximum value among them as the heuristic [6, 7]. Since so much memory was released we were able to use the extra memory for storing six different 8-8-8 partitionings illustrated in figure 5 and taking their maximum as the heuristic.

Method	Nodes	Entries	Mem	Hits
1 6-6-6-6	4,756,891,097	255,024,000	244	100
2 6-6-6-6	1,107,142,063	255,024,000	244	100
1 Gigabyte				
OD	1,434,852,411	7,178,143	125	27.7
OD+6	938,256,516	7,178,143	369	28.9
PP	856,917,588	25,071,429	656	66.4
Imp1	714,722,200	25,071,429	656	68.5
Imp2	714,722,200	4,538,015	327	16.6
6 8-8-8	198,851,450	15,638,294	505	-
8 9-9-6	175,100,719	31,199,159	754	-
2 Gigabytes				
PP	713,536,979	66,690,152	1,322	80.6
Imp1	613,844,599	66,690,152	1,322	83.3
Imp2	613,844,599	13,565,100	472	21.2
6 8-8-8	130,890,131	48,907,720	1,037	-
8 9-9-6	100,964,443	82,143,861	1,569	-

Table 1. ISPDB with 8-8-8 and 9-9-6 partitionings

5 Experimental results

We implemented all the above variations and improvements on the same random instances of the 24 puzzle used by [8]. We experimented with the 8-8-8 partitionings of figure 5 and also with a set of 9-9-6 partitionings. We used a 1.7 MHz Pentium 4 PC with one gigabyte of main memory and also with two gigabytes. The primary search was performed with IDA*.

We sorted the 50 instances from [8] in increasing order of solution length. Table 1 presents the average results on the ten ”easiest” instances. The first column indicates the variation used and the second column counts the number of generated nodes. The third column, *Entries*, is the total number of 8-8-8 (or 9-9-6) PDBs entries that were finally stored. The *Mem*, gives the total amount of memory in megabytes used for all the PDBs consulted by this variation (including the 6-6-6-6 PDB when applicable). Finally, the last column indicates the percentage of times where the 8-8-8 (9-9-6) PDB had a hit. We define a *hit* as a case where a PDB is consulted and actually had an entry for the specific pattern. This is opposed to a *miss* where that entry was not available and the simpler heuristics were consulted.

Often, a stronger heuristic consumes more time per node. Thus, the overall time improvement to solve the problem with a stronger heuristic is less than the reduction in the number of generated nodes. Nevertheless, the actual time is influenced by the effectiveness and effort devoted to the current implementation. For example, using a better hash function or sophisticated data structures for storing entries in the PDB might further improve the running time. A number of methods for reducing the constant time per node when using multiple PDBs lookups were provided by [6]. Using as many of these methods further reduces the overall time. The actual time also depends on the hardware and memory abilities of the machine used. We noted that the number of nodes

per second in all our variations was always between one to two Million. Since the nodes improvement reported below is significantly greater we decided to omit the time reports and concentrate only on the number of generated nodes. As discussed above, we can also omit the time of generating the PDBs which took between 30 to 80 minutes for our different variations. This is negligible when compared to the actual search time which was a around 18 hours on average for the random 50 instances.

The first row of table 1 uses one 6-6-6-6 partitioning. The second row is the benchmark results taken from [8] where the 6-6-6-6 partitioning was also reflected about the main diagonal and the maximum between the two was used. This reduces the search effort by a factor of 4.

In the next bunch of rows we had one gigabyte of main memory for the secondary A*. The first row (OD) is the simple case where only a single 8-8-8 partitioning (of figure 5.a) was used and the extended secondary A* search was performed on demand. In a case of a miss in a PDB, we calculated the Manhattan distance (MD) for the tiles in this particular PDB. Here, only 7,178,143 entries were generated since the open lists of the different 8-tile PDBs were stored in memory during the primary search. Note that the hit ratio here is low (27.7) as the size of the PDB is comparably small. Even this simple variation of one 8-8-8 partitioning reduced the number of generated nodes by more than a factor of three when compared to the one 6-6-6-6 partitioning.

The second row (OD+6) also generated the PDBs on demand. However, here, (and in all the successive rows) we took the maximum between the 8-8-8 and the 6-6-6-6 PDBs. This variation outperformed the one 6-6-6-6 version (line 1) by a factor of 5 and the benchmark two 6-6-6-6 version (line 2) by 18%. The next line (PP) is the preprocessing variation where the entire secondary search for each 8-tile PDB was performed a priori until memory was exhausted. Here more patterns were expanded by the secondary A* search and therefore we could load 25,071,429 values to the PDBs. This reduced the number of generated nodes to 856,917,588. Note that the hit ratio was increased to 66%. While the total number of patterns for three 8-tile sets is 129 Billion entries we stored only 25 Million (a fold factor of 4,600) and yet had the relevant value in 66% of the times.

The fourth line (Improvemnet 1) used the 6-2 partitioning instead of MD when a miss occurred. The fifth line (Improvement 2) only loaded the 8-tile values that are larger than the corresponding 6-2 partitioning. Here we can see a reduction of a factor of four in the number of stored entries. In both variations the number of generated nodes was reduced to 714,722,200. With improvement 2, the hit ratio dropped to 16.6 since many of the entries were removed as they were no larger than the 6-2 partitionings. Note that improvement 2 squeezed the PDB to a small size which enabled us to store multiple PDBs below.

In the next line full advantage of main memory was taken and six different 8-8-8 partitionings were stored. This reduced the number of nodes to 198,851,450. In the last line we used 8 9-9-6 PDBs. Here the number of nodes is 175,100,719, eight times smaller then benchmark results of two 6-6-6-6 partitionings. Here we did not report the hit ratio as it was difficult to define it for multiple lookups.

We then report similar experiments performed when we had two gigabytes of main memory for generating the PDBs. Here, more patterns were generated and the final

Method	Mem	Nodes	Ratio1	Ratio2
The 25 easiest instances				
2 6-6-6-6	-	16,413,254,279	1	1
8 9-9-6	1GB	3,788,144,197	4.33	7.37
6 8-8-8	1GB	2,897,728,901	5.66	6.59
8 9-9-6	2GB	2,339,671,729	7.01	12.85
6 8-8-8	2GB	2,037,614,978	8.05	10.25
All 50 instances				
2 6-6-6-6	-	360,892,479,671	1	1
6 8-8-8	2GB	65,135,068,005	5.54	8.85

Table 2. Results over 25 and 50 instances

PDBs were larger. Note that the hit ratio here increased to 83.3%. The number of generated nodes here was 130,890,131 for the multiple 8-8-8 PDBs and 100,964,443 for the 9-9-6 PDBs. This is approximately one order of magnitude better than the previous benchmark results for these 10 instances.

Our most successful method used six 8-8-8 and eight 9-9-6 partitionings. With this method we solved the ”easiest” 25 instances (the first 25 from the sorted list of instances) with 1GB and 2GB of memory. Results are presented in Table 2. Since the number of generated nodes in the tile puzzle is exponentially distributed it is problematic to report average results. Therefore, we report two different numbers for the improvement factor over the 6-6-6-6 benchmark. The first number (*Ratio1*) is the ratio of the total number of nodes for the 25 instances. For the second number (*Ratio 2*) we calculated the improvement factor for each instance alone and report the average over all the 25 factors. When considering the total number of generated nodes over all instances, the 8-8-8 partitioning improved the benchmark results by a factor of 8 and when considering each instance alone the 9-9-6 reduced the number of nodes by a factor of almost 13.

Finally, we solved the entire set of 50 instances from [8] with 6 8-8-8 PDBs. It seems that the effectiveness of IDPDB drops a little for the more difficult instances. This is because there are more A* patterns but the PDB has the same size and therefore the chance for *missing* a pattern from the PDB increases. Still, even in difficult instances, IDPDB managed to focus on the relevant patterns and the total number of nodes over all 50 instances was 65,135,068,005 - 5.54 smaller than pervious benchmark results. On a single instance basis the improvement factor ranged from 3.5 to 40 and averaged 8.85. It took us about a month to solve all 50 instances. To the best of our knowledge we have the best published optimal solver for this problem.

6 Summary and conclusions

.

We presented simplifications and enhancements to the instance dependent PDB method suggested by [10]. We showed that instead of the fixed mathematical set they computed, we can dynamically grow the PDB until memory is exhausted. With our

method we optimally solved random instances of the 24 puzzle by using 8-8-8 (9-9-6) disjoint partitionings. A single complete 8-tile PDB needs 43 Billion entries. We reduced the number of entries by a factor of 1000 and yet produced the state of the art performance for this problem.

For future work we intend to combine partitionings of different sizes e.g. an 8-8-8 with a 9-9-6 etc. We would also intend to build a larger hierarchy of different pattern databases each adds only relevant values to its predecessor. For example, given a 5-tile pattern database, we can use it for adding a sixth tile by only storing the contribution of this new tile to the 5-tile pattern databases. Also, a deeper mathematical analysis and a theory that unifies this work with the work of [10] should be introduced.

References

1. J. C. Cullberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
2. S. Edelkamp. Planning with pattern databases. *Proceedings of the 6th European Conference on Planning (ECP-01)*, 2001.
3. A. Felner, R. E. Korf, and Sarit Hanan. Addtive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, 22:279–318, 2004.
4. A. Felner, R. Meshulam, R. Holte, and R. Korf. Compressing pattern databases. In *Proc. AAAI-04*, pages 638–643, 2004.
5. I. Hernádvölgyi and R. C. Holte. Experiments with automatically created memory-based heuristics. *Proc. SARA-2000, Lecture Notes in Artificial Intelligence*, 1864:281–290, 2000.
6. R Holte, J. Newton, A. Felner, and D. Furcy. Multiple pattern databases. *Proc. ICAPS-04*, pages 122–131, 2004.
7. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. *Proc.AAAI-97*, pages 700–705, 1997.
8. R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.
9. M. McNaughton, P. Lu, J. Schaeffer, and D. Szafron. Memory efficient A* heuristics for multiple sequence alignment. In *Proc. AAAI-02*, pages 737–743, 2002.
10. R. Zhou and E. Hansen. Space-efficient memory-based heuristics. In *Proc. AAAI-04*, pages 677–682, 2004.
11. R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *Proc. AAAI-04*, pages 683–689, 2004.