Compressing Pattern Databases

Ariel Felner and **Ram Meshulam**
Computer Science Department
Bar-Ilan University
Ramat-Gan, Israel 92500
Email: {felner,meshulr1}@cs.biu.ac.il

Robert C. Holte
Computing Science Department
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: holte@cs.ualberta.ca

Richard E. Korf
Computer Science Department
University of California
Los Angeles, CA 90095
Email: korf@cs.ucla.edu

Abstract

A pattern database is a heuristic function stored as a lookup table which stores the lengths of optimal solutions for instances of subproblems. All previous pattern databases had a distinct entry in the table for each subproblem instance. In this paper we investigate compressing pattern databases by merging several adjacent entries into one, thereby allowing the use of pattern databases that exceed available memory in their uncompressed form. We show that since adjacent entries are highly correlated, much of the information is preserved. Experiments on the sliding tile puzzles and the 4-peg Towers of Hanoi puzzle show that, for a given amount of memory, search time is reduced by up to 3 orders of magnitude by using compressed pattern databases.

Introduction

Heuristic search algorithms such as A* and IDA* are guided by 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.

Pattern databases are heuristics in the form of lookup tables. They have proven very useful for finding lower bounds for combinatorial puzzles and other problems(Culberson & Schaeffer 1998; Korf 1997; Korf & Felner 2002).

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. A *pattern* is a state of the subproblem. The *pattern space* for a given subproblem is a state space containing all the different patterns connected to one another using the same transition rules ("operators") that connect states in the original problem. The pattern space is an abstraction of the original space in the sense that the distance between two states in the original space is greater than or equal to the distance between the corresponding patterns. A *pattern database* (PDB) stores the distance of each pattern to the goal pattern. A PDB is built by running a breadth-first search backwards from the goal pattern until the whole pattern space is spanned. A state S in the original space is mapped to a pattern S' by ignoring details in the state description that are not preserved in the subproblem. The value stored in the PDB for S', i.e., the distance in the pattern space from S' to the goal pattern, is a lower bound on the distance of S to the goal state in the original space.

The size of a pattern space is the number of patterns it contains. As a general rule, the speed of search is inversely related to the size of the pattern space used (Hernádvölgyi & Holte 2000). Larger pattern spaces take longer to generate but that is not the limiting factor. The problem is the memory required to store the PDB. In all previous studies, the amount of memory needed for a PDB was equal to the size of the pattern space, because the PDB had one entry for each pattern. In this paper, we present a method for compressing a PDB so that it uses only a fraction of the memory that would be needed to store the PDB in its usual, uncompressed form. This permits the use of patterns spaces that are much larger than available memory, thereby reducing search time. We limit the discussion to a memory capacity of 1 gigabyte, which is reasonable in today's technology. The question we address in this paper is how to best use this amount of memory with compressed pattern databases. Experiments on the sliding tile puzzles and the 4-peg Towers of Hanoi puzzle show that search-time is significantly reduced by using compressed pattern databases.

The 4-peg Towers of Hanoi Problem

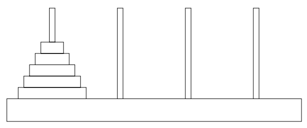

Figure 1: Five-disk four-peg Towers of Hanoi problem

The well-known 3-peg Towers of Hanoi problem consists of three pegs. The task is to transfer all the disks to the goal peg. Only the top disk on any peg can be moved and a larger disk can never be placed on top of a smaller disk. For the 3-peg problem, there is a simple recursive deterministic algorithm that provably returns an optimal solution. The 4-peg Towers of Hanoi problem (TOH4)(Hinz 1999) shown in Figure 1, is more interesting. There exists a deterministic algorithm for finding a solution, and a conjecture that it generates an optimal solution, but the conjecture remains unproven. Thus, systematic search is currently the only method guaranteed to find optimal solutions, or to verify the conjecture for problems with a given number of disks.

The domain of TOH4 has many small cycles, meaning there are many different paths between the same pair of states. For example, if we move a disk from peg A to peg B, and then another disk from peg C to peg D, applying these two moves in the opposite order will generate the same state. Therefore, any depth-first search, such as IDA*, will generate many nodes more than once and be hopelessly inefficient in this domain. Thus, only a search that prunes duplicate nodes will be efficient in this domain. We used frontier-A* (FA*), a modification of A* designed to save memory (Korf & Zhang 2000). FA* only saves the open list and deletes nodes from memory once they have been expanded. In order to keep from regenerating closed nodes, with each node on the open list the algorithm stores those operators that lead to closed nodes, and when expanding a node those operators are not used. FA* also uses a special method to reconstruct the solution path (Korf & Zhang 2000).

Pattern databases for TOH4

PDB heuristics are applicable to TOH4. Consider a 16-disk problem. We can build a PDB for the largest 10 disks by having an entry for each of the 4^{10} legal patterns of these 10 disks. The value of an entry is the minimum number of moves required to move all the disks in this 10-disk group from the corresponding pattern to the goal peg, assuming there are no other disks in the problem. Since there are exactly 4^n states for the n-disk problem, indexing this PDB is particularly easy, since each disk position can be represented by two bits, and any pattern of n disks can be uniquely represented by a binary number $2n$ bits long. Given a state of the 16-disk problem, we compute the pattern for the 10-disk group and lookup the value for this configuration in the PDB. This value is an admissible heuristic for the complete 16-disk problem because a solution to the 16-disk problem must at least move the largest 10 disks to the goal.

A similar PDB can be built for the smallest 6 disks. Values from the 10-disk PDB and the 6-disk PDB can be *added* together to get an admissible heuristic value for the complete state since a complete solution must include a solution to the largest 10 disks problem as well as to the smallest 6 disks problem. This sum is a lower bound because the two groups are disjoint and their solutions only count moves of disks within their own group. The idea that costs of *disjoint* subproblems can be *added* was first used in (Korf & Felner 2002) for the tile puzzles and inspired this formalization.

Note that a PDB based on n disks will contain exactly the same values for the largest n disks or the smallest n disks or any other n disks. The reason is that only the relative sizes of the disks matter, and not their absolute sizes. Furthermore, a PDB for n disks also contains a PDB for m disks, if $m < n$. To look up a pattern of m disks, we simply assign the $n - m$ largest disks to the goal peg, and then look up the resulting pattern in the n-disk PDB. Thus, in practice we only need a single PDB for the largest number of disks of any group of our partition. In our case, a 10-disk PDB contains both a PDB for the largest 10 disks and a PDB for the smallest 6 disks. In general, the most effective heuristic is based on partitioning the disks into the largest groups that we can, thus building the largest PDB that will fit in memory. The largest PDB we can use with a gigabyte of memory is of 14 disks. This has 4^{14} entries and needs 256 megabytes at one byte per entry. The rest of the memory is needed for the open-list of FA*.

Heuristic	Path	Avg h	Nodes	Seconds
Static 13-3	161	75.78	134,653,232	48.75
Static 14-2	161	89.10	36,479,151	14.34
Dynamic 14-2	161	95.52	12,827,732	21.56

Table 1: Results for the 16-disk problem

Given a PDB of 14 disks, there are two methods to use it for the 16-disk problem. The first is called *statically partitioned PDB*. In this method, we statically partition the disks into the largest 14 disks and the smallest 2 disks. This partition remains *static* for all the nodes of the search. The other method is called *dynamically partitioned PDB*. For each state we compute all 240 different ways of dividing the disks into groups of 14 and 2, look-up the database value for each, and return the maximum of these as the final heuristic value. Here, the exact partitioning to disjoint disks is *dynamically* determined for each state of the search on the fly. Table 1 has results for the standard initial state of the 16-disk problem. A 14-2 split is much better than a 13-3 split since the large PDB of 14 is more informed than the PDB of 13. For the 14-2 split, a dynamically partitioned heuristic is more accurate and the search generates fewer nodes (and therefore FA* requires less memory). Statically partitioned heuristics are much simpler to calculate and thus consume less overall time but generate more nodes. The static and dynamic partitioning methods can be applied in any domain where additivity of disjoint PDBs is applicable.

Compressing cliques in pattern databases

Pattern databases in all previous studies have had one entry for each pattern in the pattern space. In this section, we describe methods for compressing pattern databases by merging the entries stored in adjacent positions of the lookup table. This obviously reduces the size of the pattern database, but the key to making this successful is that the merged entries should be very similar values. While this cannot be guaranteed in general, we show in this paper that it is the case for the state spaces and indexing functions that are used in our experiments.

Suppose that K nodes of the pattern space form a clique. This means that all these nodes are reachable from each other by one edge. Thus, the PDB entries for these nodes will differ from one another by no more than 1, some will have a value N and the rest will have value $N + 1$.

In permutation problems such as TOH4 or the tile puzzles where the operators move one object at a time, cliques usually exist where all objects are in the same location except one, which is often in a nearby location. Therefore, such cliques usually have nearby entries in the PDB. If we can identify a general structure of K adjacent entries in the pattern database which store a clique of size K, we can squeeze the PDB as follows. Instead of storing K entries for the different K nodes we can have all these K nodes mapped into one entry. This can be done in the following two ways:

lossy compression - Store the minimum of these K nodes, N. The admissibility of the heuristic is preserved and the loss introduced is at most 1.

lossless compression - Store the minimum value for these K nodes, N. Store also K additional bits, one for each node of the clique, that will indicate whether the node's value is N or $N+1$. This version will preserve the entire knowledge of the large PDB but will usually require less memory.

The existence of cliques in the pattern space is domain dependent. Furthermore, when cliques exist, their adjacency in the PDB depends on the indexing function used. Finally, in order for this technique to be applicable there must exist an efficiently computable indexing function into the compressed PDB. We do not assert that these conditions can be satisfied in all domains, but we believe they hold quite broadly, at least for the permutation-type domains that we study here. Note that with lossy compression, any block of nearby entries can be compressed and admissibility is still kept. With cliques, however, we are guaranteed that the loss of data will be at most 1.

For TOH4, the largest clique is of size 4. Consider a PDB based on a subproblem of P disks, for example, the 10-disk PDB described above. Assume that the location of the largest $P-1$ disks is fixed and focus on the smallest disk which can be placed in 4 different locations. These 4 states form a clique of size 4 since the smallest disk can move among these 4 states in only 1 move. Thus, we can store a PDB of P disks in a table of size 4^{P-1} (instead of 4^P) by squeezing the 4 states of the smallest disk into one entry. If the PDB is built as a multi-dimensional array with P indices where the last index corresponds to the smallest disk, then the only difference between these 4 states is their last index with the position of the smallest disk. Thus, they are stored in 4 adjacent entries.

In the compressed PDB, we will have $P-1$ indices for the largest $P-1$ disks and only one entry for the smallest disk instead of 4 entries in the original database. Lossy compression would store the minimum of these 4 entries and lose at most 1 for some of the entries. Alternatively, lossless compression can store 4 additional bits in each entry of the compressed database which will indicate for each location of the smallest disk whether the value is N or $N+1$.

This idea can be generalized to a set of K nodes with a diameter of D, i.e, each pair of nodes within the set have at least one connecting path consisting of D or fewer edges (Note that a clique is a special case where $D=1$). We can compress the set of nodes into only one entry by taking the minimum of their entries and lose at most D. Alternatively, for the lossless compression we need an additional $K \times log(D+1)$ bits to indicate the exact value. If the size of an entry is M bits then it will be beneficial (memory-wise) to use this compression mechanism for sets of nodes with diameter of D as long as $log(D+1) < M$.

TOH	h(s)	Avg h	D	Nodes	Time	Mem
16_0^{14}	119	89.10	0	36,479,151	14.34	256
16_1^{14}	118	88.55	1	37,964,227	14.69	64
16_2^{14}	116	87.74	3	40,055,436	15.41	16
16_3^{14}	114	86.53	5	44,996,743	16.94	4
16_4^{14}	110	84.80	9	45,808,328	17.36	1
16_5^{14}	107	82.91	13	61,132,726	23.78	1/4
16_{1s}^{14}	119	89.10	0	36,479,151	15.87	96

Table 2: Solving 16-disks with a pattern of 14 disks

For example for TOH4 this generalization applies as follows. We fix the position of largest $P-2$ disks and focus on the 16 different possibilities of the two smallest disks. These possibilities form a set of nodes with diameter 3, and it is easy to see that they are placed in 16 adjacent entries. Thus, we can squeeze these 16 entries to one entry and lose at most 3 for any state. Alternatively, we can add $2 \times 16 = 32$ bits to the one byte for the entry (for a total of 5 bytes) and store the exact values. This is instead of 16 bytes in the simple uncompressed database.

Experiments on the 4-peg Towers of Hanoi

As a first step we compressed the 14-disk PDB to a smaller size. Define a *compression degree of* z to denote a PDB that was compressed by storing all different positions of the smallest z disks in one entry given that the rest of the disks are fixed. The amount of memory saved with a lossy compression degree of z is 4^z. We define $TOHx_z^y$ to denote a 4-peg Towers of Hanoi problem with x disks that was solved by a PDB with y disks compressed by a degree of z. For the 16-disk problem we define our PDBs by statically dividing the disks into two groups. The largest fourteen disks (disks 1-14) define the 14-disk PDB. The smallest two (disks 15 and 16) are in their own group and have a separate, uncompressed PDB with $4^2 = 16$ entries. To compute the heuristic for a state, the values for the state from the small PDB and the 14-disk PDB are added. Notice the difference between a 14-2 split where two separate PDBs are built (14 and 2) and a PDB of 14 compressed by a degree of 2 where the specific 14-disk PDB is compressed.

Table 2 presents results of solving the standard initial state (where all disks are initially located on one peg) of the 16-disk problem, which has an optimal solution of 161 moves. Different rows of the table correspond to different compression degrees of the 14-disk PDB. All but the last row represent lossy compression. The first row of Table 2 is for the complete 14-disk database with no compression while row 6 has a compression degree of 5. The third row for example, has a compressing degree of 2. In that case, the PDB only contains 4^{12} entries which correspond to the different possibilities of placing disks 1-12. For each of these entries, we take the minimum of all the 16 possibilities for disks 13 and 14 and have only one entry for them instead of 16.

The last column gives the size of the PDB in megabytes. The most important result here is that when compressing the PDB by a factor of $4^5 = 1024$, most of the information is not

TOH	Type	Avg h	Nodes	Time	M
17_0^{14}	static	90.5	>393,887,912	>421	256
17_0^{14}	dynam	95.7	238,561,590	2501	256
17_1^{15}	static	103.7	155,737,832	83	256
17_2^{16}	static	123.8	17,293,603	7	256
18_2^{16}	static	123.8	380,117,836	463	256

Table 3: Solving larger versions of TOH4

lost. Such a large compression increased the search effort by less than a factor of 2 for both the number of generated nodes and for the time to solve a problem.

The last row represents lossless compression of the full 14-disk database by a degree of 1 where we stored 1 additional bit for each position of the smallest disk in the 14-disk group (#14). This needed 12 bits per 4 entries instead of the 32 bits in the uncompressed PDB (row 1). While the number of generated nodes is identical to row 1, the table clearly shows that it is not worthwhile to use lossless compression for TOH4 since it requires more time and more memory than lossy compression by a degree of both 1 and 2.

The maximum possible loss of data for lossy compression with a degree of z – the diameter D – is presented in column D of Table 2. This is the length of the optimal solution for a problem with z disks, because the two farthest states with z disks are those that have all disks on one different peg. The *Avg h* column with the average heuristic on all possible entries shows that on average, the loss was half the maximum. Note that the loss for the heuristic of the standard initial state is shown in the $h(s)$ column is exactly the maximum, D.

Larger versions of the problem

With lossy compression we also solved the 17- and 18-disk problems. The shortest paths from the standard initial state are 193 and 225 respectively. Results are presented in Table 3. An uncompressed *statically partitioned* PDB of the largest 14 disks and the smallest 3 disks cannot solve the 17-disk problem since memory is exhausted before reaching the goal after 7 minutes (row 1). With an uncompressed PDB of 14 disks we were only able to solve the 17-disk problem with *dynamically partitioned* PDB (row 2).

The largest database that we could precompute when performing a breadth-first search backwards from the goal configuration was for 16 disks. Our machine has 1 gigabyte of memory. Thus, when tracking nodes with a bit map we need 4^{16} = 4 gigabits, half the size of our memory. Given the same amount of memory as the full 14-disk database, 256MB, we solved the 17-disk problem in 83 seconds with a 15-disk PDB compressed by a degree of 1, and in 7 seconds with a 16-disk PDB compressed by a degree of 2. This is an improvement of at least 2 orders of magnitude compared to row 1. The improvement is almost 3 orders of magnitude compared to the dynamic partitioning heuristic of the 14-disk PDB row 2. While a PDB of 16 disks compressed by a degree of 2 consumes exactly the same amount of memory as an uncompressed PDB of 14 disks it is much more informed as it includes almost all the data about 16 disks. With a 16-disk database compressed by a degree of 2 we were also able to solve the 18-disk problem in a number of minutes. Note that (Korf 2004) solved this problem in 16 hours when using the delayed duplicate detection (DDD) breadth-first search algorithm.

The system that we described here is able to find a shortest path for any possible initial state of TOH4. However, one can do much better if a shortest path is only needed from the standard initial state where all disks are located on one peg (Hinz 1999). For this special initial state, one can only search half way to an intermediate state where the largest disk can move freely from its initial peg to the goal peg. In such a state all the other $n - 1$ disks are distributed over the other two pegs. To complete the solution we apply the moves to reach such an intermediate state in the reverse order and interchange the initial and goal pegs. Based on this symmetry, (Korf 2004) was able to obtain a shortest path for the standard initial state with a DDD breadth-first search for TOH4 with up to 24 disks. However, this symmetry doesn't apply to arbitrary initial and goal states where a complete search must be performed to the goal state as our system does. Furthermore, his system needed tens of gigabytes and took 19 days.

Experiments on the Sliding Tile Puzzles

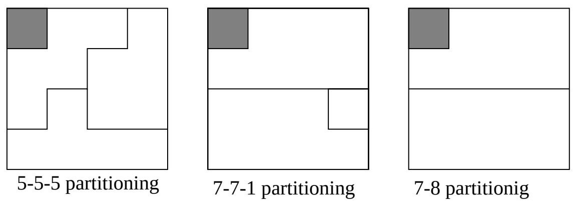

Figure 2: Different disjoint databases for the Fifteen Puzzle

The best existing method for solving the tile puzzles uses disjoint pattern databases (Korf & Felner 2002). The tiles are partitioned into disjoint sets (subproblems) and a PDB is built for each set. The PDB stores the cost of moving the tiles in the given subproblem from any given arrangement to their goal positions. If for each set of tiles we only count moves of tiles from the given set, 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. Figure 2 shows a 5-5-5, a 7-7-1 and a 7-8 disjoint partitioning of the 15-puzzle.

Taking advantage of simple heuristics

In many domains there exists a simple heuristic, such as Manhattan distance (MD) for the sliding tile puzzles, that can be calculated very quickly. In these domains, a PDB can store just the addition above that heuristic. During the search we add values from the PDB to the simple heuristic.

For the tile puzzles we can therefore store just the addition above MD which correspond to conflicts and internal interactions between the tiles. These conflicts come in units of 2 moves, since if a tile moves away from its *Manhattan-distance path* it must return to that path again with a total of 2 additional moves to its MD. Compressing PDBs can greatly benefit from this idea. Consider a pair of adjacent entries in the PDB. While their MD is always different by one, the addition above the MD is most of the time exactly the same. Thus, much of the data is preserved when taking the minimum. In fact, for the 7-7-1 partition in Figure 2, we have found that more than 80% of the pairs we grouped stored exactly the same value.

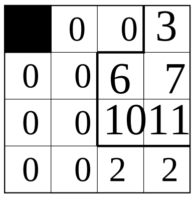

Figure 3: One pattern of the Fifteen puzzle

For example, consider the subproblem of $\{3,6,7,10,11\}$ shown in Figure 3. Suppose that all these tiles except tile 6 are located in their goal position and that tile 6 is not in its goal position. The values in Figure 3 written in location x correspond to the number of steps above MD that the tiles of the subproblem must move in order to properly place tile 6 in its goal location given that its current location is x. For example, suppose that tile 6 is placed below tile 10 or tile 11. In that case tile 6 is in linear conflict with tiles 10 or 11 and one of them must move at least two more moves above MD. Thus we write the number 2 in these locations. For other locations we write 0 as no additional moves are needed. Locations where other tiles are placed are treated as *don't-care* as tile 6 cannot be placed at these locations. Note that most adjacent positions have the same value.

For TOH4, one can create a simple heuristic (similar to MD) concentrating on the number of moves that each disk must move. However, this heuristic is very inaccurate and proved ineffective in conjunction with PDBs.

Storing PDB for the Tile Puzzles

While a multi-dimensional array of size 4^P is the obvious way to store a PDB for TOH4, there are two ways to store PDBs for the tile puzzles. Suppose for example, that we would like to store a PDB of 7 tiles for the 15-puzzle. There are $16 \times 15 \times \ldots \times 10$ different possible configurations to place these 7 tiles in 16 locations. A simple way would be to store a 7-dimensional array. This will need 16^7 different entries but the access time is very fast. The other idea is to have a simple one-dimensional array of exactly $16 \times 15 \times \ldots \times 10$ entries, but use a complex mapping function in order to retrieve the exact entry of a given permutation. This is done as follows. The first tile can be located in 16 positions. The next tile can only be positions in 15 locations etc, The mapping function should calculate all these possibilities and return a unique value for each configuration. We refer to the first option as *simple mapping* and the second method as *packed mapping*.

For the simple mapping, there are 16 different entries for the last tile which correspond to the 16 different possible locations of the 15-puzzle. We divide the 16 locations into 8 pairs: (0,1), (2,3) ... (14,15). Instead of storing 16 entries for the last tile, we can store just 8 entries, one for each of these pairs. Thus, the size of the PDB will be $16^6 \times 8$ which is half the size of the original database. Since a legal move can only move one tile to a nearby location the largest clique in this puzzle is of size 2. By pairing neighboring locations of the last tile we take advantage of such cliques[1].

For the packed mapping it is a little more complicated. If the PDB is based on k tiles, there are only $16 - k + 1$ entries for the last tile. For example, if $k = 7$ there are only 10 legal positions to place the last tile. If we use the same pairing mechanism described above then we can compress these 10 entries to 8 entries. This method will only be effective if the number of tiles is considerably smaller than half the size of the puzzle. For example, in the 24-puzzle, it will be efficient to compress 6 tiles even with the packed mapping.

H	P	C	Nodes	Time	Mem	Av h
1 7-8	P	-	136,288	.081	576,575	44.75
1+ 7-8	P	-	36,710	.034	576,575	45,63
1 7-7-1	P	-	464,977	.232	57,657	43.64
1 7-7-1	S	-	464,977	.058	536,870	43.64
1 7-7-1	S	l	565,881	.069	268,435	43.02
1 7-7-1	S	s	487,430	.070	268,435	43,59
2 7-7-1	S	l	147,336	.021	536,870	43.98
2+ 7-7-1	S	l	66,692	.016	536,870	44.92

Table 4: 15-puzzle results.

Results on the Fifteen puzzle

Table 4 presents results on the 15-puzzle. All the values in the table are the average over the 1000 random initial states that were used in (Korf & Felner 2002). We used a 2.4GH pentium 4 with 1 gigabyte of main memory. The first column defines the heuristic. **1 7-7-1** means that we used only one 7-7-1 partitioning. **2 7-7-1** means that we used 2 different 7-7-1 partitionings and took their maximum as the heuristic. A **+** means that we also took the same partitioning and reflected it about the main diagonal. The second column indicates whether we used simple mapping (S) or packed mapping (P) and the next column indicates whether we used no compression (-), lossy compression (l) or lossless compression (s). The next columns present the number of nodes generated by IDA*, the average time in seconds, the amount of memory in K bytes (at one byte per entry) and the average initial heuristic. The time needed to precompute the PDB is traditionally omitted since one only needs to precompute it once and then solve as many problem instances as needed.

The first two rows present the same results of a 7-8 partitioning that was obtained by (Korf & Felner 2002) but on our current machine. Note that while (Korf & Felner 2002) report a running time of 0.027 seconds, exactly the same

[1]There are rare cases which only occur in 2.5% of the cases, where the above pairs are not a clique. This is due to the location of the blank tile (details are omitted). However, taking a minimum of any two values (even if they are not a clique) is always admissible.

software took 0.034 seconds on our current machine which has a faster CPU. The reason is that hardware abilities of a given machine such as cache performance, memory-cpu data exchange rate and internal hardware structure have a lot of influence on the actual overall running time.

The third and fourth rows present the results of the 7-7-1 database from Figure 2 but with the different mapping systems. Notice the time versus memory tradeoff here.

The fifth row gives the results of the 7-7-1 PDB after compressing each pair of entries described above into one entry with lossy compression. While the size of the PDB was reduced by a factor of 2, the overall effort was increased by no more than 20% in both the number of generated nodes and in the overall time. The next row presents results of the same 7-7-1 partitioning when we used lossless compression. While the number of generated node decreased by 15% from the lossy compression, the overall time increased a little. This is due to the additional constant time complexity of the bit handling of the lossless compression[2]. We have also tried to compress 4 adjacent entries of the PDB into one but this proved inefficient on the 15-puzzle as much data was lost.

The last two rows show the benefit of compression. The seventh row presents results when we took two different 7-7-1 partitionings, compressed them and took their maximum. This configuration uses the same amount of memory as one 7-7-1 partitioning of row 4 but solves the problem almost three times faster. It is also faster and uses less memory than the 7-8 partitioning (row 1). Finally, the last row also computes the reflection about the main diagonal of these two compressed databases and takes the maximum of the 4 different partitionings. This further reduced the running time and we now solve a problem in only 16 milliseconds. This is faster by a factor of two, and uses less memory, than the best 7-8 partitioning used in (Korf & Felner 2002) (row 2).

Results on the 24-puzzle

The best existing heuristic for 24-puzzle is the 6-6-6-6 partitioning and its reflection about the main diagonal from (Korf & Felner 2002). We compressed the same 6-6-6-6 partitioning and found that like the 15-puzzle the lossy compression generated nearly 20% more nodes. However, with adding another 6-6-6-6 partitioning we could not achieve any significant reduction in the overall time. Due to geometrical attributes of the puzzle, the 6-6-6-6 partitioning and its reflection from (Korf & Felner 2002) are so good that adding another 6-6-6-6 partitioning (even without compressing anything) only achieves a small reduction in node generations which is not compensated by the time overhead.

We have also tried A 7-7-5-5 partitioning (and its refection) on this domain which could be stored in 1 gigabyte of memory if the 7-tile databases are compressed. Even without compressing, the number of generated nodes was not significantly different from the best 6-6-6-6 partitioning. The 6-6-6-6 partitioning of (Korf & Felner 2002) is probably the best 4-way partitioning of the 24-puzzle.

The only way to obtain a speedup in this domain is to compress larger databases such as an 8-8-8 partitioning. However, we need much more than 1 gigabyte to generate this database with breadth-first search, and that is beyond the scope of the current set of experiments.

[2]The reason that the number of generated nodes was not identical to the complete 7-7-1 partitioning is because as described above, in rare cases, two nearby entries are not a clique and differ by more than one. Thus, data is lost even with the lossless compression that we used.

Conclusions and Future Work

We introduced a method that better utilizes memory by compressing PDBs and showed applications on the tile puzzles and on TOH4. In both domains significant compression was achieved, allowing larger pattern spaces to be used and search time to be considerably reduced. For the 15-puzzle and for TOH4 with arbitrary initial states we have the state of the art solvers. Our experiments confirm that given a specific amount of memory, M, it is better to compress larger PDBs into M entries than to use an uncompressed PDB with M entries. We have also showed two methods (i.e., static and dynamic) for partitioning disjoint patterns.

The memory limits imposed by using ordinary breadth-first search to generate very large pattern databases that are subsequently compressed might be overcome by using delayed duplicate detection (DDD)(Korf 2004). This is a method for performing best-first search which stores the open and/or closed lists on the disk. With DDD, one can run a breadth-first search on pattern spaces that are much larger than the available memory. Values from this breadth-first search can then be compressed to a database that can fit in memory. For example, one can run a breadth-first search for subproblems of 8 tiles for the 24-puzzle and then compress the values into 1 gigabyte of memory.

Future work will continue these ideas as follows. Advanced data structures (like a trie for the tile puzzles) might perform better than simple tables as they will have more flexibility with regards to what entries to compress. Another interesting approach would be to feed a learning system like a neural network with values from the PDB. Also, other ideas for having a selective PDBs which only keep important values can be developed.

References

Hinz, A. 1999. The Tower of Hanoi, *Algebras and Combinatorics: Proceedings of ICAC'97*. 277–289

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

Hernádvölgyi, I. T., and Holte, R. C. 2000. Experiments with automatically created memory-based heuristics. *Proc. SARA-2000, Lecture Notes in Artificial Intelligence* 1864:281–290.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134:9–22.

Korf, R., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment,. *Proc. AAAI-2000*, 910–916.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. *Proc. AAAI-97*, 700–705.

Korf, R. E. 2004. Best-first search with delayed duplicate detection *Proc. AAAI04*, San-Jose, Ca.