1.6-Bit Pattern Databases

Teresa M. Breyer and **Richard E. Korf**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{tbreyer,korf}@cs.ucla.edu

Abstract

We present a new technique to compress consistent pattern databases without loss of information by storing the heuristic estimate modulo three, requiring only two bits per entry, or in a more compact representation only 1.6 bits. This enables us to store a pattern database with four or five times as many entries in the same amount of memory as an uncompressed pattern database. We compare both methods to the best existing compression methods for the Top-Spin puzzle, Rubik's cube, the 4-peg Towers of Hanoi Problem, and the 24-puzzle. For the Top-Spin puzzle and Rubik's cube we also compare our best implementation to the respective state of the art solvers. This compression technique is most useful where methods for lossy compression fail, for example where patterns mapping to adjacent entries in the pattern database are not reachable from each other by one move, such as in the Top-Spin puzzle and Rubik's cube.

Introduction

Heuristic search algorithms, including A* (Hart, Nilsson, & Raphael 1968), IDA* (Korf 1985), Frontier A* (Korf *et al.* 2005), and Breadth-First Heuristic Search (BFHS) (Zhou & Hansen 2006) use a cost function f to prune nodes. f assigns to each state n a cost $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the shortest path found so far from the start state to state n, and $h(n)$ is the heuristic estimate of the lowest cost to get from n to a goal state. If $h(n)$ never overestimates the lowest cost from n to a goal state, it is *admissible* and optimality of the solution is guaranteed. If $h(n)$ never decreases by more than the cost of the operator applied, it is *consistent*. Formally, $h(n)$ is consistent if $h(n) \leq c(n,m) + h(m)$ for all neighboring states n and m, where $c(n,m)$ is the cost of the operator to get from n to m. Most naturally occurring heuristics are consistent, but lossy compression, maximizing over several pattern databases, or using duality generate inconsistent heuristics (Zahavi *et al.* 2007; Felner *et al.* 2007; Holte *et al.* 2006; Zahavi *et al.* 2008).

For many problems, a heuristic evaluation function can be calculated before the search and stored in a lookup table called a *pattern database* (PDB) (Culberson & Schaeffer 1998). For example, for the Towers of Hanoi problem we choose a subset of the discs, the *pattern discs*, and ignore the positions of all non-pattern discs. For each possible configuration of the pattern discs we store the minimum number of moves required to solve this smaller Towers of Hanoi problem in a lookup table. In general, a *pattern* is a projection of a state from the original problem space onto the pattern space. In our example the pattern would be the state consisting of only the pattern discs. The projection of the goal state is called the *goal pattern*. For each pattern the minimum number of moves required to reach the goal pattern in the pattern space is stored in the PDB. PDBs are constructed through a backward breadth-first search from the goal pattern in the pattern space. For each pattern the entry in the PDB is the depth at which it is first generated. Under certain conditions it is possible to sum values from several PDBs without overestimating the solution cost (Korf & Felner 2002). For the Towers of Hanoi problem we can partition all discs into disjoint groups and construct a PDB for each of these groups. Since each operator moves only one disc, it only affects discs in one PDB. In general, if there is a way to partition all variables into disjoint sets of pattern variables so that each operator only changes variables from one set of pattern variables, we call the resulting PDBs *additive*, and such a set of PDBs are called *disjoint*.

This paper focuses on lossless compression of PDBs. In general, the more variables used as pattern variables in a PDB, the more entries the PDB has, and the more accurate the resulting heuristic estimate will be. If we can losslessly compress PDBs, we can fit PDBs with more entries in memory and thus solve problems with fewer node expansions than when using the same amount of space for an uncompressed PDB. This research applies to all problem spaces where operators have unit cost and are reversible.

Examples of Pattern Databases

The Top-Spin Puzzle

The (n,k)-Top-Spin puzzle consists of a circular track holding n tokens, numbered from 1 to n. The goal is to arrange the tokens in order. The tokens can be slid around the track, and there is a turnstile that can flip the k tokens it holds. In the most commonly used encoding, an operator is a shift around the track by 0 to $n-1$ positions followed by a re-

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

versal of the k tokens in the turnstile. Using this encoding, this puzzle can be implemented as a cyclic buffer where each operator reverses k consecutive tokens (Felner *et al.* 2007). In the physical puzzle the tokens have to be rotated into the turnslide first.

We construct PDBs for this problem by only considering a subset of the tokens, the *pattern tokens*, and for each pattern storing the number of flips required to arrange the pattern tokens in order relative to each other. Unlike in the Towers of Hanoi problem the non-pattern tokens are present, but indistinguishable. Each operator reverses k tokens, and these tokens could belong to different pattern sets. Hence it is not trivial to construct additive PDBs. Yang *et al.* (2008) first suggested a technique to construct more general additive PDBs for this problem.

Rubik's Cube

We construct PDBs for Rubik's cube by only considering a subset of the cubies. Korf (1997) first solved random instances of Rubik's cube using the maximum of three different PDBs, one based on the 8 corner cubies, and the other two based on 6 edge cubies each. As in the Top-Spin puzzle, for this problem it is not trivial to construct additive PDBs. For Rubik's cube Yang *et al.* (2008) were not able to report any improvements using their more general additive PDBs.

The Sliding-Tile Puzzles

The 15-puzzle is a 4×4 sliding-tile puzzle. The minimum number of moves needed to get a subset of tiles, the *pattern tiles*, to their goal positions is a lower bound on the total number of moves required to solve the puzzle, and thus an admissible heuristic function. For each possible configuration of pattern tiles and the blank, this value is stored in a PDB. If we only count moves of the pattern tiles when constructing the PDBs, we can use several disjoint groups of pattern tiles and sum the values from each of these individual PDBs to get an admissible heuristic function (Korf & Felner 2002). To save memory, instead of storing one heuristic value for each position of the blank and each configuration of the pattern tiles, it is common practice to only store the minimum over all positions of the blank for each configuration of the pattern tiles, i.e., the additive disjoint PDBs are in practice usually compressed by the position of the blank, making them inconsistent (Zahavi *et al.* 2007).

Related Work

Mod Three Breadth-First Search

Cooperman & Finkelstein (1992) introduced this method to compactly represent problem space graphs. A function, also called a *perfect hash function*, is used to map each state to a unique index in a hashtable. This table stores two bits for each index. Initially all states have a value of three, labeling them as not yet generated, and the root has a value of zero. A breadth-first search of the graph is performed and the hashtable is used as the open list during search. While searching the graph, for each state the depth at which it is first generated is stored modulo three, thus the values zero to two label states that have been generated. When the root is expanded, its children are assigned a value of one. In the second iteration the whole hashtable is scanned, all states with a value of one are expanded, and states generated for the first time are assigned a value of two. In the third iteration the complete hashtable is scanned again, all states with value two are expanded, and states generated for the first time are assigned a value of zero (three modulo three). In the following iteration states with value zero are expanded. Thus the root will be re-expanded, but no new states will be generated from it. Basically, each state that has been expanded will be re-expanded once every three iterations, but previously expanded states will not generate any new states. When no new states are generated in a complete iteration, the search ends and all reachable states have been expanded and assigned their depth modulo three.

Given this hashtable and any state, one can easily determine the depth at which the state was first expanded as well as a path back to the root. First the state is expanded, then an operator that leads to a state at one depth shallower (modulo three) is chosen and that state is expanded. This process is repeated until the root is generated. The number of steps it took to reach the root is the depth of the state, and all states expanded form a path from the root to the original state.

Compressed Pattern Databases

Felner *et al.* (2007) performed a comprehensive study of methods to compress PDBs. They concluded that, given limited memory, it is better to use this memory for compressed than for uncompressed PDBs. Most of their methods are lossy, resulting in inconsistent heuristics. They range from compressing a PDB of size M by a factor of c by using a simple function that maps c patterns to one compressed entry, to more compex methods leveraging specific properties of problem spaces. To map c patterns to one entry, one can use the index into the uncompressed PDB and divide it by c, to get the index in the compressed database. Alternatively one can take the original index modulo M/c, the size of the compressed PDB. Each entry stores the minimum over the heuristic estimates of all c patterns mapped to it.

Felner *et al.* (2007) introduced only one method for lossless compression. It is based on *cliques*. In the context of PDBs a *clique* is defined as a set of patterns all of which are reachable from each other by one move. Given a clique of size q, one can store the minimum value d and q additional bits, one bit for each pattern in the clique. This bit is set to zero if the pattern's heuristic value is d or to one if the heuristic value is $d + 1$. More generally, a set of z nodes, where any pair of nodes is at most r moves apart, can be compressed by storing the minimum value d and an additional $\lceil \log_2 (r + 1) \rceil$ bits per pattern. For each pattern these bits store a number between zero and r, where zero stands for heuristic value d, one for $d + 1$, up to r for $d + r$. In the 4-peg Towers of Hanoi problem, patterns differing only by the position of the smallest disc form a clique of size four, all one move away from each other. Patterns differing only by the positions of the smallest two discs form a set of 16 nodes at most three moves apart and can be compressed using one byte for the minimum value d and two bits for each pattern storing whether the heuristic value is d, $d+1$, $d+2$ or $d+3$.

In the sliding-tile puzzle graph, the maximum clique size is only two or one edge. All tiles are fixed except for one tile which can be in any one of two adjacent positions. These two states map to adjacent entries in the PDB and thus can be efficiently compressed. In the Top-Spin puzzle graph, the only cliques are edges as well, but these edges connect states that differ by more than one pattern token, since each move flips k tokens.

Cliques can also be used for lossy compression. For a clique of size q, instead of storing the minimum value d and q additional bits indicating whether each pattern's heuristic value is d or $d + 1$, one can just store d. This introduces an error of at most one move. Similarly a set of z nodes, where any pair of nodes is at most r moves apart, can be compressed by storing the minimum value introducing an error of at most r moves. For example, if we compress a PDB for the 4-peg Towers of Hanoi by ignoring the position of the smallest pattern disc, we can compress by a factor of four and introduce an error of at most one move. If we ignore the positions of the smallest two discs, we can compress by a factor of 16 and introduce an error of at most three moves.

Another lossless compression method used in planning are symbolic PDBs (Edelkamp 2002; Ball & Holte 2008). Symbolic PDBs use binary decision diagrams (BDDs) or algebraic decision diagrams (ADDs) to store a PDB and for some search spaces significantly reduce the memory needed to store the PDB. Results of compression rates of over $100 : 1$ are reported. But sometimes they can require even more memory than the original PDB, for example for the 15-puzzle. Finally, another lossy compression method uses artificial neural networks and learning to compress PDBs (Samadi *et al.* 2008). This method is also able to compress by large factors, but it requires the uncompressed PDBs to be build and stored in memory, which is a limiting factor of this technique.

Two-Bit Pattern Databases

Uncompressed PDBs usually assign one byte for each entry, which is sufficient as long as the maximum heuristic estimate does not exceed 255. This is the case for all of the problem spaces we consider here. We reduce this without loss of information to two bits per pattern state, so we are able to compress by a factor of four, by using Cooperman & Finkelstein's (1992) modulo three breadth-first search to construct and store losslessly compressed consistent PDBs. In particular, the two-bit PDB holds the heuristic estimates modulo three. Even if the original heuristic estimate of a pattern required more than a byte in the uncompressed PDB, two bits per pattern state would suffice using our compression. In general, if the uncompressed PDB used N bits per state we are able to compress by a factor of $N/2$. This method requires no particular properties of the search space, except for a consistent heuristic estimate and operators that have unit edge cost and are reversible. We can store a single PDB or a set of disjoint PDBs as two-bit PDBs and we can use these compressed PDBs with any heuristic search algorithm, including A*, IDA*, Frontier A*, and BFHS.

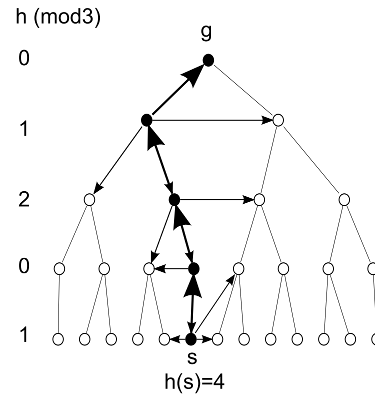

Figure 1: Two-Bit PDB Lookup of Start State

Algorithm

First we construct the two-bit PDB using Cooperman & Finkelstein's (1992) modulo three breadth-first search in the pattern space with the projection of the goal state onto the pattern space as the root. To avoid re-expansion of nodes once every three levels, we keep a first-in first-out queue of open states on disk. This is a standard method used for constructing PDBs. We use the modulo three hashtable to check whether a state has been generated, and to store the depth modulo three at which a state is first generated, which is the heuristic estimate modulo three of all preimages of the state in the original problem space. When the search is completed, this hashtable is our two-bit PDB.

We can use this two-bit PDB to solve any particular problem instance using any heuristic search algorithm. Figure 1 shows an example. First, we determine the heuristic estimate of the start state. We start by projecting the start state onto our pattern space which gives us the start pattern s. Secondly we expand the start pattern in the pattern space. At least one operator will lead to a neighbor one level (modulo three) closer to the goal pattern g. Then any one of these patterns is chosen for expansion in the pattern space. In our example the start pattern has heuristic value one (modulo three) and we chose one of the two children with heuristic zero (modulo three) for expansion next. We keep track of the number of steps taken. This process is repeated until the goal pattern is generated. The number of steps it took to reach the goal pattern in the pattern space is the heuristic estimate of the start state in the original problem space. In our example the start state has heuristic estimate four.

We store the heuristic estimate of the start pattern together with the start state in the open list or on the stack, depending on the search algorithm used. When we generate a new state, and it is not in our open or closed list in case of A*, Frontier-A* or BFHS, we have to calculate the heuristic estimate of the child. Figure 2 demonstrates how a two-bit PDB lookup is done. We take the heuristic estimate of the parent pattern p modulo three, or alternatively we look up the entry of the parent pattern in the PDB, and we look up the entry of the child pattern c in the PDB. Comparing these two modulo three values will tell us whether the child pattern is

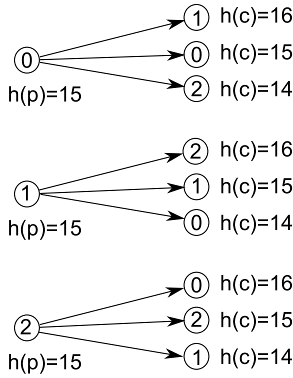

Figure 2: Two-Bit PDB Lookup during Search

a level closer to the goal, further away or at the same level as the parent. Figure 2 shows all possible lookups in a two-bit PDB for a parent pattern with heuristic estimate 15. We get the child's heuristic estimate by adding one to the parent's heuristic estimate, subtracting one from the parent's heuristic estimate, or assigning the child pattern the same heuristic estimate as the parent. Finally, we store the child state with the child's heuristic estimate in the open list or on the stack if it does not exceed the upper bound.

In case of multiple PDBs, we construct each PDB using Cooperman & Finkelstein's (1992) modulo three breadth-first search. To solve a problem instance, we determine the heuristic estimates of the start state for each PDB by back-tracking from the start pattern to the goal pattern in the pattern space. We store the start state's heuristic estimates together with the state in the open list or on the stack. During search we incrementally calculate each heuristic estimate of a newly generated state from its parent the same way we did for a single PDB. In case of additive PDBs, we only need to calculate the additive heuristic estimate that has the modified variables as pattern variables. The other heuristic estimates carry over unchanged from the parent state.

Using Less Than Two Bits

For a consistent PDB, only three values are required to store the depth modulo three for each pattern. Using two bits per state allocates four distinct values per pattern, but the fourth value was only used while constructing the PDB and thus is not required to store or use the PDB. Instead of two contiguous bits per state, we can compress the PDB using base three numbers. Two mod three value can be encoded as one of 9 different values, three mod three values as one of 27 different values, etc. While constructing the PDB we can use a separate table, which tells us whether a pattern state has been generated or not. A perfect hash function assigns each pattern one bit in this table, its *used bit*. Initially all used bits are set to zero. When a state is generated its used bit is set to one. After constructing the PDB we can discard this table of used bits. The resulting PDB requires $\log_2 3 \approx 1.58$ bits per state and is the most efficient method for lossless compression of these particular values. Cooperman & Finkelstein (1992) also mention this improvement.

#	Comp.	$h(s)$	Generated	Time	Size
1	None	10.53	43,607,741	12.04	495
2	Two Bit	10.53	43,607,741	12.83	123
3	Mod 4	9.84	86,184,716	24.75	123
4	1.6 Bit	10.53	43,607,741	13.57	99
5	Mod 5	9.63	95,952,807	27.57	99

Table 1: Solving the (17,4) Top-Spin Puzzle using a 9-Token PDB

Theoretically we would need a total of $n \cdot \log_2 3$ bits to store a PDB with n entries or $\lceil n \cdot \log_2 3/8 \rceil$ bytes. But accessing the correct values per pattern gets computationally expensive, because it involves integer division and modulo operators on very large numbers. In particular a PDB would be represented as a single multi-word number. In our experiments later in this paper we use PDBs that take one gigabyte of memory or more, thus when using 1.58 bits per pattern we would have to work with one number taking approximately one gigabyte and extract our heuristic estimates (modulo three) from that number. Alternatively, we decided to fit as many patterns as possible in each byte. Each byte represents $2^8 = 256$ different values, the largest power of three number that can be stored in one byte is $3^5 = 243$, thus we can fit five mod three values in one byte. Compared to using one byte per pattern this allows us to compress by a factor of five and uses $8/5 = 1.6$ bits per pattern, which is very close to the optimal 1.58 bits per state determined before. Even with maximal compression only 20 entries could be encoded in one four-byte word, or 40 entries in one eight-byte word, thus encoding five entries per byte is just as efficient. Accessing an entry in the 1.6 bit PDB is still slightly more expensive than accessing an entry in a two-bit PDB, because it involves integer division by 5 and a modulo operator to determine the correct byte and the index into the byte. Then for each possible value of a byte and for each index into the byte we can store the actual modulo three heuristic estimate in a lookup table. This table has exactly $243 \cdot 5$ entries. In contrast to 1.6-bit PDBs, when using two-bit PDBs simple shift and bitwise and operators suffice.

Experimental Results

The Top-Spin Puzzle

As mentioned earlier, in the Top-Spin puzzle problem space graph the largest cliques are of size two, but these cliques do not correspond to compression of single tokens, since each move flips k tokens. But mapping patterns to the same compressed entry by applying integer division or modulo to their index can always be used for compressing. Felner *et al.* (2007) compared these two hash functions and established that using the modulo function performed best for Top-Spin. Their explanation for this counterintuitive result was that the distance between two states that are similar, such as two states differing only by the locations of two tokens, is greater than the distance between two random states, since each operator flips several tokens at the same time. We compare Felner *et al.*'s modulo hash function to our methods. Both compression methods use the modulo operator, ours stores

the heuristic values modulo three, theirs applies the modulo operator to the hash function. To avoid confusion, we will call our methods *two-bit*, and *1.6-bit PDBs*, and their compression method *modulo compression*.

Table 1 has experimental results on the $(17, 4)$ Top-Spin problem, which has 17 tokens and a turnstile that flips four tokens. We used IDA* with a PDB consisting of tokens 1 through 9. Our experiments are averaged over $1,000$ random initial states, generated by a random walk of 150 moves, because not all permutations of the $(17, 4)$ puzzle are solvable (Chen & Skiena 1996). The average solution depth of these random initial states is 14.90. The first column gives the type of compression used. The second column gives the average heuristic value of the initial state. The third column gives the average number of nodes generated. The fourth column gives the average time in seconds required to solve a problem, and the last column gives the size of the PDB in megabytes. The rows are ordered by the size of the PDB starting with the largest one. The first row gives results using the uncompressed PDB, which uses one byte per entry. The second row uses the same PDB stored modulo three using two bits per entry. The third row uses modulo compression by a factor of four using the same amount of memory as our two-bit PDB. The fourth row uses even less memory storing five entries per byte using our 1.6-bit PDB. The next row uses the same amount of memory using modulo compression by a factor of five.

One can see that our lossless methods perform almost as well as the uncompressed PDB, but use only a quarter or a fifth of memory to store the PDB. Obviously the same number of nodes are generated. The two-bit PDB also performs almost equally well time wise. It does not add a significant time overhead for the slightly more complex PDB lookup. For the 1.6-bit PDB, there is a slightly larger time overhead for the more expensive PDB lookup. In comparison, the modulo compression by a factor of four generates and expands about twice as many nodes and takes twice as much time as the uncompressed PDB, because basically random states are mapped to the same entry in the hashtable and only the minimum of their heuristic values can be stored. Modulo compression by a factor of five performs even worse.

Table 2 has similar results on the same data set with a PDB consisting of the tokens 1 through 10. We could not run experiments with the uncompressed PDB, because it would require approximately 4 gigabytes of RAM.

Finally, in Table 3 we use the property that a PDB for tokens 0 to 10 is also a PDB for tokens 1 to 11, 2 to 12, etc. Thus from one PDB we can actually get up to 17 different PDB lookups for the $(17, 4)$ Top-Spin problem. In Table 3 we compare our best implementation, which uses the maximum over 8 regular lookups in our 10-token PDB, to Felner *et al.*'s (2005) best implementation, which uses the maximum over 4 regular and 4 dual lookups in an uncompressed 9-token PDB as well as bpmx cutoffs. For an explanation of this algorithm we refer the reader to Felner *et al.*'s paper. Both implementations use IDA* on the same $1,000$ random instances. The first column gives the type of compression used. The second column gives the number of regular ('r') and dual ('d') lookups and the presence of bpmx cutoff ('c').

#	Comp.	$h(s)$	Generated	Time	Size
1	Two Bit	11.43	5,786,954	1.89	990
2	Mod 4	10.76	10,969,452	3.59	990
3	1.6 Bit	11.43	5,786,954	1.99	792
4	Mod 5	10.58	11,945,920	3.91	792

Table 2: Solving the (17,4) Top-Spin Puzzle using a 10-Token PDB

#	Comp.	Heur.	$h(s)$	Generated	Time	Size
1	Two Bit	8r+0d	12.37	11,103	0.016	990
2	None	4r+4d+c	11.53	76,932	0.080	495

Table 3: Solving the (17,4) Top-Spin Puzzle using a 10-Token PDB using Dual Search

The third column gives the average heuristic value of the initial state. The fourth column gives the average number of nodes generated. The second to last column gives the average time in seconds required to solve a problem, and the last column gives the size of the PDB in megabytes. One can see that our algorithm is approximately four times faster than Felner *et al.*'s (2005) dual search for this particular problem size but using twice the memory.

Overall for different problem sizes and different amounts of memory either dual search or two-bit PDBs might perform better. Our experiments strongly suggest that two-bit or 1.6-bit PDBs outperform dual search when compression enables us to store a PDB using more pattern variables in memory than when using an uncompressed PDB with dual search. We showed this to hold for the $(17, 4)$ puzzle and two gigabytes of memory.

Rubik's Cube

Felner *et al.* (2007) did not include any experiments on Rubik's cube. Thus we compare their general compression methods applying division and modulo to the index, which we will call division and modulo compression, to our two-bit and 1.6-bit PDBs.

For our first set of experiments we used the maximum over three PDBs, one based on the 8 corner cubies, and the others based on 7 edge cubies each. These PDBs have such low values, that four bits per state suffice to begin with. Thus with our method we can only compress by a factor of two and 2.5 respectively. Table 4 has experimental results averaged over the ten random initial states published by Korf (1997). Their average solution depth is 17.50. The columns are the same as in Table 1. The first row gives results using the uncompressed PDBs, which use four bits per entry. The second row uses the same PDBs stored modulo three using two bits per entry. The third row uses even less memory storing five entries per byte using our 1.6-bit PDBs. We delayed comparing to modulo and division compression until our second set of experiments with larger PDBs, because with these weaker PDBs solving all ten instances took too long. One can see that our two-bit PDBs expand the same number of nodes and add little time overhead compared to the uncompressed PDBs. The last three rows of Table 4 be-

#	Comp.	$h(s)$	Generated	Time	Size
1	None	9.1	102,891,122,415	32,457	529
2	Two-Bit	9.1	102,891,122,415	32,113	265
3	1.6-Bit	9.1	102,891,122,415	35,190	212
4	$8\text{-}8_{10}\text{-}8_{10}$	9.1	105,720,641,791	36,385	529
5	Dual	9.1	65,932,517,927	27,150	529
6	Two-Bit (8-7-7-7-7)	9.1	64,713,886,881	27,960	529

Table 4: Solving Korf's ten random initial states of Rubik's Cube using a 8 corner cubie and 2 7 edge cubie PDBs

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	9.5	26,370,698,776	11,290	1,239
2	Div 2	9.3	56,173,197,862	25,917	1,239
3	Mod 2	9.3	58,777,491,012	27,577	1,239
4	1.6-Bit	9.5	26,370,698,776	12,309	991
5	Div 2.5	9.1	68,635,164,093	33,838	991
6	Mod 2.5	9.0	77,981,222,043	35,976	991
7	Dual	9.1	65,932,517,927	27,150	529
8	Two-Bit (8-8-8-8-8)	9.7	14,095,769,007	8,667	1,239

Table 5: Solving Korf's ten random initial states of Rubik's Cube using a 8 corner cubie and 2 8 edge cubie PDB

low the line have slightly different experimental results. The first row below the line uses the same amount of memory as the uncompressed PDBs, but instead of using two PDBs based on 7 edge cubies each, it uses two PDBs based on 8 edge cubies each compressed to the size of a PDB based on 7 edge cubies using division compression by a factor of 10. The second row uses the original uncompressed PDBs, but it uses the regular and dual lookup for both 7-edge-cubie PDBs (Zahavi *et al.* 2008), thus the maximum of a total of five lookups. We believe that this is currently the best solver for Rubik's cube in the heuristic search community. The last row also used five lookups, but instead of using two dual and two regular lookups it uses four regular lookups in the 7 edge cubies PDBs using geometric symmetries. One can see that five regular lookups perform just as well as a combination of regular and dual lookups. Also, for Rubik's cube there seems to be no advantage from compressing a larger PDB to the size of a smaller PDB.

For our second set of experiments we used the maximum over three PDBs, one based on 8 corner cubies, and two based on 8 edge cubies. Due to geometrical symmetries we only need to store one of these 8-edge-cubie PDBs. The uncompressed PDB based on 8 edge cubies does not fit in two gigabytes of memory, so we can only compare the compressed PDBs here. Similar to Table 4, Table 5 has experimental results averaged over Korf's (1997) ten random initial states. The first row uses our two-bit PDBs. The second and third rows use modulo and division compression by a factor of two using four bits per entry and so using the same amount of memory as our two-bit PDBs. The fourth row uses 1.6-bit PDBs and the fifth and sixth row use the same amount of memory using division and modulo compression by a factor of 2.5. One can see that modulo and division compression by a factor of two expand twice as many nodes and take twice as much time as using the uncompressed PDBs, while our two-bit PDBs expand the same number of nodes and add little time overhead. Unlike in the Top-Spin puzzle, for Rubik's cube modulo and division compression perform equally poorly. In the first row below the line in Table 5 we also give experimental results for the best existing solver using dual lookups. But since it uses uncompressed PDBs, we can only give results using the PDBs using only 7 edge cubies. Again we compared against five regular lookups in the second row, four of which are in the two-bit 8 edge cubie PDB. Here one can see that with two gigabytes of memory our best implementation beats the best existing implementation by a factor of four but using more than twice as much memory. We also tried using more than four regular lookups in the 8 edge cubie PDB, but there is only a reduction in number of nodes expanded, time-wise we could not achieve any further speed-up.

Summarizing, for Rubik's Cube our two-bit and 1.6-bit PDBs are the best compressed PDBs. Unlike in other problem spaces, we showed that given limited memory it is not necessarily better to use this memory for compressed than for uncompressed PDBs, in particular when using lossy compression. Also, with our implementation we beat the fastest solver currently available which uses regular and dual lookups by a factor of four.

The 4-peg Towers of Hanoi Problem

Lossy compression methods using cliques and their generalization have proved very effective for the 4-peg Towers of Hanoi (Felner *et al.* 2007). Compressing by several orders of magnitude still preserves most information. Even with additive PDBs it is most efficient to construct a PDB with as many discs as possible and compress it to fit in memory and to use the remaining discs in a small uncompressed PDB. The state of the art for this problem (Korf & Felner 2007) used a 22 disc PDB compressed to the size of a 15 disc PDB by ignoring the positions of the 7 smallest discs. This introduces an error of at most 25 moves, the number of moves required to solve a problem consisting of 7 discs. We limit our experiments to PDBs for which the hashtable of used bits and the compressed PDB fit in two gigabytes of memory. Implementing a more involved search algorithm using magnetic disk exceeds the scope of this paper. The used bit table for the 16 disc Towers of Hanoi problem requires 4^{16} bits or 512 megabytes and is the largest one that will fit in our memory. The maximum heuristic value for this PDB is 161, thus the uncompressed PDB uses one byte per entry. Using Felner *et al.*'s method, ignoring the positions of the smallest discs, we can compress by any multiple of four. For the Towers of Hanoi problem an exponential memory algorithm that detects all duplicates is the algorithm of choice. We use breadth-first heuristic search (BFHS) in our experiments. BFHS searches the problem space in breadth-first order but uses f-costs to prune states that exceed a certain threshold. Usually an iterative deepening approach is applied and one starts with the heuristic estimate of the start state as the threshold and increases it by one in every iter-

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	162	18,006,252	5.75	1,024
2	16_1	161	19,472,851	6.18	1,024
3	1.6-Bit	162	18,006,252	6.11	820
4	16_2	159	21,819,725	6.65	256
5	16_3	157	25,579,490	8.13	64

Table 6: Solving the 17 Discs Towers of Hanoi Problem using a 16 Disc PDB

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	164	355,856,206	333	1,024
2	16_1	163	373,045,641	355	1,024
3	1.6-Bit	164	355,856,206	336	820
4	16_2	161	400,505,833	387	256
5	16_3	159	443,154,284	443	64

Table 7: Solving the 18 Discs Towers of Hanoi Problem using a 16 Disc PDB

ation. Here we only need to perform one iteration with the optimal solution cost as an upper bound, because the presumed cost of an optimal solution is known (Frame 1941; Stewart 1941).

Experimental results using this PDB on the 17 disc problem with different levels of compression are shown in Table 6. The columns are the same as in Table 1. The uncompressed PDB would have required four gigabytes of memory, and so is not feasible. The first row has results using our two-bit PDB, the second row uses the same amount of memory compressing the same PDB by a factor of four by ignoring the smallest disc. The third row uses our 1.6-bit PDB. The fourth row ignores the two smallest discs and the last row ignores the three smallest discs. Table 7 has similar results for the 18 disc problem. In both tables one can see that very little information is lost using lossy compression and there is no real gain from storing the complete 16 disc PDB without loss of information in memory.

In short, because of cliques and generalized cliques lossy compression is so powerful that we can only achieve a slight improvement with lossless compression. Also, lossy compression allows compressing by any multiple of four, while we can only compress by a factor of at most five.

The Sliding-Tile Puzzles

For the 24 puzzle, the state of the art is Dual IDA* (Zahavi *et al.* 2008) with a 6-6-6-6 partitioning and its reflection as the heuristic function. This 6-6-6-6 partitioning is considered the best 4-way partitioning for this puzzle and no major improvements using compression were reported by Felner *et al.* (2007). To use this partitioning with dual search a total of eight 6-tile PDBs need to be stored, while regular IDA* only requires two 6-tile PDBs.

For permutation problems there are two ways to map patterns to indices in the PDB. In sparse mapping each pattern variable is used as a separate index into the PDB. This obviously wastes some memory, because entries with two or more equal indices do not correspond to actual states, since no two tiles can occupy the same location. The advantage is that these indices can be calculated very efficiently. In contrast to sparse mapping there is compact mapping, in which there is a one-to-one mapping between entries and patterns. This method does not waste any memory, but computing the index becomes more expensive. For IDA* the above 6-6-6-6 partitioning uses two 6-tile PDBs requiring 243 megabytes memory when using compact mapping and 466 megabytes when using sparse mapping. Even though it uses more memory, sparse mapping is faster than compact mapping, even when using Korf & Schultze's (2005) mapping algorithm, which is linear in the number of pattern tiles.

#	Comp.	Generated	Time	Size
1	Two-Bit	48,398,394,783	203	1,155
2	1.6-Bit	48,398,394,783	208	924
3	sparse	114,988,269,664	202	466
4	compact	114,988,269,664	395	243

Table 8: Solving the 24 Puzzle using a 6-6-6-6 Tile PDB

As mentioned earlier, the additive disjoint PDBs for the sliding-tile puzzles are usually compressed by the position of the blank. Unfortunately, this makes them inconsistent, so we cannot apply our method directly. The uncompressed 6-6-6-6 partitioning which has one entry for each configuration of the pattern tiles and the blank requires over four gigabytes of memory, but using our two-bit PDBs or 1.6-bit PDBs and compact mapping it fits in 1155 megabytes or 925 megabytes respectively. Sparse mapping would require almost three gigabytes of memory and hence is infeasible.

Table 8 has experimental results. For simplicity, we only used one 6-6-6-6 partitioning and IDA* to see if our technique shows any promise on this problem space. Results are averaged over the easiest ten instances from Korf & Felner's (2002) 50 random instances. The columns have the same meaning as in Table 1, only time is in minutes. The first row has results using two bits per state for the losslessly compressed 6-tile PDB, and the second row uses our 1.6-bit PDB, both with compact mapping. Both of these compressions take into account the position of the blank. The third row gives results using the classic 6-6-6-6 partitioning and sparse mapping. The last row uses the same PDB with compact mapping. The latter two both compress by the position of the blank. One can see that lossless compression expands about 57% fewer nodes but time wise sparse mapping is still slightly faster and uses less memory. Thus we did not even implement using the maximum over one 6-6-6-6 partitioning and its reflection and we did not run experiments on the more difficult benchmark problems.

Conclusions and Future Work

We have introduced a lossless compression method for PDBs which can store a consistent heuristic in just 1.6 or alternatively two bits per state. We have applied this compression method to four different problems, the Top-Spin puzzle, Rubik's cube, the sliding-tile puzzle and the Towers of Hanoi problem, combined with IDA* as well as with BFHS. For the Top-Spin puzzle, 1.6 and two-bit PDBs perform sig-

nificantly better than general compression methods applying division and modulo to the index. Depending on the memory available our method even outperforms dual search. For Rubik's cube we showed that these general compression methods fail, and our method is the best compression method. With our best implementation we beat the fastest solver currently available, which uses regular and dual lookups, by a factor of four when limited to two gigabytes of memory. For the Towers of Hanoi problem lossy methods are so good that we could not improve on them. For the 24 puzzle we were able to reduce the number of nodes expanded, but we could not improve on the classic 6-6-6-6 partitioning time wise and required more memory. In general our lossless compression is useful for problems that do not allow lossy compression using cliques or their generalization or where adjacent entries in the PDB are not highly correlated.

Future work includes using this compression method with dual search, which results in an inconsistent heuristic. For dual lookups the heuristic values have to be read from the two-bit PDBs the same way as for the start state.

It is also possible to generalize our method to inconsistent PDB heuristics. Zahavi *et al.* (2007) defined the inconsistency rate of a heuristic h and an edge $e = (m, n)$ as $|h(n) - h(m)|$. A heuristic with a maximum inconsistency rate of k over all edges can be represented using $i = \lceil log_2(2 \cdot (k+1)) \rceil$ bits per entry. Applying an operator can increase or decrease the heuristic estimate by a value of up to k, so $2k+1$ values are required and one more value is used during construction of the PDB. If i is smaller than the number of bits required per state in the uncompressed PDB there is a memory gain from our compression.

Our method can also be extended to some problems where operators do not have unit edge cost and are not reversible, for example to sequence alignment (Hohwald, Thayer, & Korf 2003; Zhou & Hansen 2003). PDBs constructed from sub-alignments of sequences with a maximum edge cost of k can be represented using $i = \lceil log_2(2 \cdot (k+1)) \rceil$ bits per entry. Again if i is smaller than the number of bits required per state in the uncompressed PDB there is a memory gain from compressing modulo $2k+1$. This method could also be combined with external-memory PDBs (Zhou & Hansen 2005) reducing the time to read blocks of the PDB from disk.

Acknowledgment

This research was supported by NSF grant No. IIS-0713178. Thanks to Satish Gupta and IBM for providing the machine these experiments were run on.

References

Ball, M., and Holte, R. C. 2008. The compression power of symbolic pattern databases. In *ICAPS-08*, 2–11.

Chen, T., and Skiena, S. S. 1996. Sorting with fixed-length reversals. *Discrete Applied Mathematics* 71:269–295.

Cooperman, G., and Finkelstein, L. 1992. New methods for using cayley graphs in interconnection networks. *Discrete Applied Mathematics* 37:95–118.

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

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *AIPS-02*, 274–283.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *IJCAI-05*, 103–108.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *JAIR* 30:213–247.

Frame, J. S. 1941. Solution to advanced problem 3918. *American Mathematical Monthly* 48:216–217.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Hohwald, H.; Thayer, I.; and Korf, R. E. 2003. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *IJCAI-03*, 1239–1245.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artif. Intell.* 170(16-17):1123–1136.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artif. Intell.* 134(1-2):9–22.

Korf, R. E., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg Towers of Hanoi problem. In *IJCAI-07*, 2324–2329.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *AAAI-05*, 1380–1385.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *J. ACM* 52(5):715–748.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *IJCAI-85*, 1034–1036.

Korf, R. E. 1997. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI-97*, 700–705.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases with learning. In *ECAI-08*, 495–499.

Stewart, B. 1941. Solution to advanced problem 3918. *American Mathematical Monthly* 48:217–219.

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

Zahavi, U.; Felner, A.; Schaeffer, J.; and Sturtevant, N. R. 2007. Inconsistent heuristics. In *AAAI-07*, 1211–1216.

Zahavi, U.; Felner, A.; Holte, R. C.; and Schaeffer, J. 2008. Duality in permutation state spaces and the dual search algorithm. *Artif. Intell.* 172(4-5):514–540.

Zhou, R., and Hansen, E. A. 2003. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *ICTAI-03*, 427–434.

Zhou, R., and Hansen, E. A. 2005. External-memory pattern databases using structured duplicate detection. In *AAAI-05*, 1398–1405.

Zhou, R., and Hansen, E. A. 2006. Breadth-first heuristic search. *Artif. Intell.* 170(4):385–408.