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 pattern databases to provide consistent heuristics without loss of information. We store the heuristic estimate modulo three, requiring only two bits per entry or in a more compact representation, only 1.6 bits. This allows us to store a pattern database with more entries in the same amount of memory as an uncompressed pattern database. These compression techniques are most useful when lossy compression using cliques or their generalization is not possible or where adjacent entries in the pattern database are not highly correlated. We compare both techniques 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. Under certain conditions, our best implementations for the Top-Spin puzzle and Rubik's cube outperform the respective state of the art solvers by a factor of four.

Introduction

Heuristic search algorithms, including A* (Hart, Nilsson, and Raphael 1968), IDA* (Korf 1985), Frontier A* (Korf et al. 2005), and Breadth-First Heuristic Search (BFHS) (Zhou and Hansen 2006) use a heuristic function h to prune nodes. $h(n)$ estimates a lowest cost to get from node n to a goal state. If h never overestimates this cost, it is *admissible* and optimality of the solution is guaranteed. If $h(n) \leq k(n,m) + h(m)$ for all states n and m, where $k(n,m)$ is the cost of a shortest path from n to m, h is *consistent*. Most naturally occurring heuristics are consistent, but lossy compression, randomizing among several pattern databases, or using duality (Zahavi et al. 2007; Felner et al. 2007; Zahavi et al. 2008) generates inconsistent heuristics.

For many problems, a heuristic evaluation function can be precomputed and stored in a lookup table called a *pattern database* (PDB) (Culberson and 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 other 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. For problems with unit operator costs, PDBs are constructed through a backward breadth-first search from the projection of the goal state, the *goal pattern*, in the pattern space. A perfect hash function maps each pattern to one entry in the PDB where the depth at which it is first generated is stored. This is exactly the minimum number of moves required to reach the goal pattern in the pattern space.

During search, we get the heuristic estimate of a state by projecting it onto the pattern space, and then using the perfect hash function to retrieve the pattern's entry in the PDB. Under certain conditions, it is possible to sum values from several PDBs without overestimating the solution cost (Korf and Felner 2002). For the Towers of Hanoi problem, we can partition all discs into disjoint sets and construct a PDB for each of these sets. In general, if there is a way to partition all state variables into disjoint sets of pattern variables so that each operator only changes variables from one set, we can add the resulting heuristic estimates admissibly. We call the resulting PDBs *additive* and such a set of PDBs *disjoint*.

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 therefore 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.

Other Examples of Pattern Databases

The Sliding-Tile Puzzles

We construct PDBs for these puzzles by only considering a subset of the tiles, the *pattern tiles*, and for each pattern storing the number of moves required to get the pattern tiles to their goal positions. Unlike in the Towers of Hanoi problem, non-pattern tiles are present, but indistinguishable. If we only count moves of pattern tiles, we can use disjoint sets of pattern tiles to generate disjoint additive PDBs (Korf and Felner 2002). To save memory, instead of storing one heuristic value for each position of the blank and each configuration of the pattern tiles, Korf and Felner only stored the minimum over all positions of the blank for each pattern, resulting in an inconsistent heuristic (Zahavi et al. 2007).

The Top-Spin Puzzle

The (n, k) Top-Spin puzzle consists of a circular track holding n tokens, numbered 1 to n. The goal is to arrange the tokens so that they are sorted in increasing 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 common encoding, an operator is a shift around the track followed by a reversal of the k tokens in the turnstile.

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. The non-pattern tokens are present, but indistinguishable. Each operator reverses k tokens, and these tokens could belong to different pattern sets. Hence these PDBs are not additive.

Rubik's Cube

We construct PDBs for Rubik's cube by only considering a subset of the cubies, usually the eight corner cubies or a set of edge cubies. For each pattern, we store the number of rotations required to get the *pattern cubies* in their goal positions and orientations (Korf 1997). As in the Top-Spin puzzle, these Rubik's cube PDBs are not additive.

Related Work

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. Their methods range from compressing a PDB of size M by a factor of c by mapping c patterns to one compressed entry using any function from the set $\{1, \cdots, M\}$ to the set $\{1, \cdots, M/c\}$, to more complex methods leveraging specific properties of problem spaces. To map c patterns to one entry, one can use the index in the uncompressed PDB and divide it by c, to get the index in the compressed PDB. Alternatively, one can take the original index modulo M/c. Each entry stores the minimum heuristic estimates of all c patterns mapped to it.

Felner et al. (2007) introduced a method for lossy compression based on *cliques*. Here, a *clique* is a set of patterns reachable from each other by one move. For a clique of size q, one can just store the minimum value 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 d. This introduces an error of at most r moves. In the 4-peg Towers of Hanoi problem, patterns differing only by the position of the smallest disc form a clique of size four. Therefore, we can compress PDBs for this problem by a factor of four by ignoring the position of the smallest disc. Patterns differing only by the positions of the smallest two discs form a set of 16 nodes at most three moves apart. Therefore, we can compress by a factor of 16 by ignoring the positions of the smallest two discs. This introduces an error of at most three moves. In the sliding-tile puzzle graph, the maximum clique size is only two. All tiles are fixed except for one specific tile which can be in any one of two adjacent positions. These two states are mapped to adjacent entries in the PDB and therefore can be efficiently compressed by dividing the index by two. In the Top-Spin puzzle graph, the only cliques are of size two as well, but these two states differ by more than one pattern token, since each move flips k tokens.

Cliques can also be used for lossless compression. 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 i between zero and r, where i represents heuristic value $d + i$.

Mod Three Breadth-First Search

Cooperman and Finkelstein (1992) introduced this method to compactly represent problem space graphs. A *perfect hash function* and its inverse are used to map each state to a unique index in a hashtable which stores two bits for each index and vice versa. For example, for the 4-peg Towers of Hanoi problem with n discs, the hash function assigns each state an index consisting of $2n$ bits, two bits for the location of each disc. Initially, all states have a value of three, labeling them as not yet generated, and the initial state 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 modulo three at which it is first generated is stored. Therefore, 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. Therefore, the root will be re-expanded, but all child states no longer have a value of three. Consequently, no new states will be generated from it. 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 determine the depth at which that state was first generated, 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.

Two-Bit Pattern Databases

Uncompressed PDBs often assign one byte or four bits to each entry, which is sufficient as long as the maximum

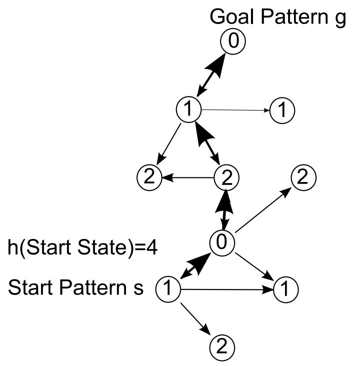

Figure 1: Two-bit PDB lookup of start state

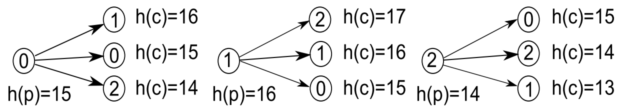

Figure 2: Two-bit PDB lookup during search

heuristic estimate does not exceed 255 or 15, respectively. This is the case in all of our domains. We reduce this without loss of information to two bits per entry, so we are able to compress by a factor of four or two. In general, if the uncompressed PDB used N bits per entry, we are able to compress by a factor of $N/2$. This method only requires unit edge cost reversible operators and consistent heuristics.

Constructing Two-Bit Pattern Databases

We use Cooperman and Finkelstein's (1992) modulo three breadth-first search in the pattern space with the goal pattern as the root to construct our PDB. When the search is completed, the hashtable is our two-bit PDB. We can use this two-bit PDB to solve any particular problem instance.

Search Using Two-Bit Pattern Databases

First, we determine the heuristic estimate of the start state. Figure 1 shows an example. We begin with the start pattern s, the projection of the start state onto the pattern space. We expand s in the pattern space. At least one operator will lead to a pattern one level (modulo three) closer to the goal pattern g, the projection of the goal state onto the pattern space. Then, any one of these patterns one level closer to g is chosen for expansion next. In our example, s has heuristic value one (modulo three), and we choose the child with heuristic value zero (modulo three). This process is repeated until g is generated. We keep track of the number of steps taken. The number of steps it took to reach g is the heuristic estimate of the start state in the original problem space. In Figure 1, the start state has heuristic estimate four. We store the start state together with its heuristic estimate in the open list or on the stack, depending on the search algorithm used. When a new state is generated, we calculate its heuristic estimate. We get its heuristic estimate by adding or subtracting one from the parent's heuristic estimate or by assigning the child pattern the same heuristic estimate as its parent. Figure 2 demonstrates how a two-bit PDB lookup is done. We take the heuristic estimate modulo three of the parent pattern p, and we look up the entry of the child pattern c in the PDB. Comparing these two modulo three values tells us whether c is a level closer to g, further away or at the same level as p. Finally, we store the child state and its heuristic estimate in the open list or on the stack and continue searching.

Using Less Than Two Bits

Only three values are required to store the heuristic estimates modulo three. Using two bits per entry allocates four distinct values per pattern, but the fourth value is only used while constructing the PDB. Instead of two contiguous bits per pattern, we can compress the PDB using base three numbers. Two modulo three values can be encoded as 1 of 9 different values, three modulo 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 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 pattern is generated, its used bit is set to one. After constructing the PDB, we can discard this table. The resulting PDB requires $\log_2 3 \approx 1.58$ bits per entry and is the most efficient method for lossless compression if there is no further structure to the data. Cooperman and Finkelstein (1992) also mention this improvement.

Theoretically we would need a total of $\lceil (n \cdot \log_2 3)/8 \rceil$ bytes to store a PDB with n entries. However, 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 one single multi-word number and we would have to extract our heuristic estimates (modulo three) from that number. Alternatively, we decided to fit as many patterns as possible in each byte. Since each byte represents $2^8 = 256$ different values, the largest base three number that can be stored in one byte is $3^5 = 243$, and therefore we can fit five modulo 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. Even with maximal compression only 20 entries could be encoded in one four-byte word, or 40 entries in one eight-byte word. Therefore, 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. It involves integer division by 5 to find the correct byte and a modulo operator to determine which value to extract from the byte. Then, for each possible value of a byte and for each of the five encoded values, we store the actual modulo three heuristic estimate in a lookup table. This table has $243 \cdot 5 = 1215$ entries. In contrast, for two-bit PDBs shift and bitwise operators suffice.

Inconsistent Heuristics

It is also possible to compress inconsistent PDBs by storing the heuristic values modulo some number. Zahavi et al. (2007) defined the inconsistency rate (IRE) of a heuristic h and an edge $e = (m, n)$ as $|h(n) - h(m)|$. A heuristic with a maximum IRE over all edges of k can be compressed using

#	Comp.	$h(s)$	Generated	Time	Size
1	None	10.53	43,607,741	12.25	247
2	Two Bit	10.53	43,607,741	12.83	123
3	Mod 2	10.16	55,244,961	16.10	123
4	1.6 Bit	10.53	43,607,741	13.57	99
5	Mod 2.5	9.97	62,266,443	18.46	99

Table 1: Solving the (17,4) Top-Spin puzzle using a 9-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	247
3	Two Bit	4r+4d+c	12.39	10,188	0.631	990

Table 2: Solving the $(17, 4)$ Top-Spin puzzle using a 10-token two-bit PDB or, a uncompressed 9-token PDB

$i = \lceil log_2(2k + 1))\rceil$ bits. Applying an operator can increase or decrease the heuristic estimate by a value of up to k, so $2k + 1$ values are required. As long as i is smaller than the number of bits required per state in the uncompressed PDB, there is a memory gain from compressing modulo $2k + 1$.

Experimental Results

The Top-Spin Puzzle

Felner et al. (2007) established mapping patterns to the same compressed entry by applying the modulo function to their index as the best compression method for Top-Spin. We compare their modulo hash function to our method. Both compression methods use the modulo operator. Ours stores the heuristic values modulo three, while 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. These PDBs have such low values that four bits per state suffice. Therefore, with our method, we can only compress by a factor of 2 and 2.5, respectively. 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 and Skiena 1996). The average solution depth of these states is 14.90. Furthermore, we were limited to two gigabytes of memory. 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 first row uses the uncompressed PDB using four bits per entry. The second row uses our two-bit PDB. The third row uses the same amount of memory using modulo compression by a factor of 2. The fourth row uses even less memory using our 1.6-bit PDB. The last row uses the same amount of memory using modulo compression by a factor of 2.5.

Our compressed PDBs generate the same number of nodes as the uncompressed PDB, but use only 50% and 40% of the memory, respectively. The two-bit PDB performs almost equally well time-wise. For the 1.6-bit PDB, there is a slightly larger time overhead for the more expensive PDB lookup. In comparison, modulo compression by a factor of 2 generates about 25% more nodes and takes 25% more time than the uncompressed PDB because virtually random states are mapped to the same entry in the PDB, which stores only the minimum of their heuristic values. Modulo compression by a factor of 2.5 performs even worse.

In our second set of experiments, we use a two-bit PDB consisting of tokens 1 through 10. We could not run experiments with the uncompressed PDB because it would require approximately two gigabytes. We use the property that a PDB for tokens 1 to 10 is also a PDB for tokens 2 to 11, 3 to 12, etc. Therefore, from one PDB we can actually get up to 17 different PDB lookups for the $(17, 4)$ Top-Spin problem. Table 2 compares our best implementation in the first row, which uses IDA* and the maximum over 8 regular lookups in a 10-token two-bit PDB, to Felner et al.'s (2005) best implementation in the second row, 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. The columns are the same as in Table 1, except for the second column, which gives the number of regular ('r') and dual ('d') lookups and the presence of bpmx cutoffs ('c'). One can see that our algorithm is four times faster than Felner et al.'s (2005) dual lookups for this particular problem size but uses four times as much memory. Adding more lookups does not reduce the time to solve a problem any further but only the number of nodes expanded.

In the third row we combined dual lookups, which result in an inconsistent heuristic, with two-bit PDBs. Thus, we had to recompute the heuristic value from scratch for every dual lookup using the same technique as for the start state. Even though the number of nodes expanded is slightly less than in row one, the dual lookups are too time consuming.

Overall, two-bit PDBs perform better than dual lookups under certain conditions and vice versa. Our experiments strongly suggest that two-bit PDBs outperform dual lookups when a two-bit PDB using more pattern variables can be stored in the available memory, but the uncompressed PDB for dual lookups cannot. We showed this to hold for the $(17, 4)$ puzzle with 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. Korf (1997) first solved random instances of Rubik's cube using IDA* and the maximum of three PDBs, one 8-corner-cubie, and two 6-edge-cubie PDBs.

For our first set of experiments we used the same three PDBs as Korf (1997), except that we used seven instead of six edge cubies. These PDBs have such low values that four bits per state suffice. Therefore, with our method we can

#	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	8-8_{10}-8_{10}	9.1	105,720,641,791	36,385	529
5	Dual	9.1	65,932,517,927	27,150	529
6	(8-7-7-7-7)	9.1	64,713,886,881	27,960	529

Table 3: Solving Korf's ten initial states of Rubik's cube using a 8-corner-cubie and two 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 4: Solving Korf's ten initial states of Rubik's cube using a 8-corner-cubie and two 8-edge-cubie or, with dual lookups, two 7-edge-cubie PDBs

only compress by a factor of 2 and 2.5, respectively. Table 3 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 our two-bit PDBs. The third row uses our 1.6-bit PDBs. One can see that our two-bit PDBs expand the same number of nodes and add no time overhead compared to the uncompressed PDBs. We delay comparing to modulo and division compression until our second set of experiments with larger PDBs because solving all ten instances took too long with these weaker PDBs. The last three rows of Table 3 below the line have slightly different experimental results. The fourth row uses the same amount of memory as the uncompressed PDBs, but instead of using two 7-edge-cubie PDBs, it uses two 8-edge-cubie PDBs each compressed to the size of a 7-edge-cubie PDB using division compression by a factor of ten as well as the 8-corner-cubie PDB. The fifth row uses the original uncompressed PDBs, including the 8-corner-cubie PDB, but it uses a regular and a dual lookup for both 7-edge-cubie PDBs (Zahavi et al. 2008). Hence, it uses the maximum of a total of five lookups. We believe that this is currently the best optimal solver for Rubik's cube. The last row also uses five lookups, but instead of a dual and a regular lookup it uses two regular lookups using geometric symmetries in each of the two uncompressed 7-edge-cubie PDBs and one lookup in the 8-corner-cubie PDB. One can see that five regular lookups perform just as well as a combination of regular and dual lookups. Also, there seems to be no advantage from compressing a larger PDB to the size of a smaller PDB when using lossy compression.

For our second set of experiments we used the maximum over three PDBs, the same 8-corner-cubie PDB, and two 8-edge-cubie PDBs. Due to geometrical symmetries we only need to store one of these 8-edge-cubie PDBs. The uncompressed 8-edge-cubie PDB does not fit in two gigabytes of memory, so we can only use it when it is compressed. Similar to Table 3, Table 4 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 2 using four bits per entry. They use 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 2 expand twice as many nodes and take twice as much time as our two-bit PDBs. In the first row below the line in Table 4 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 7-edge-cubie PDBs. 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 an improvement in number of nodes expanded, not in running time.

Summarizing, our two-bit and 1.6-bit PDBs are the best known compressed PDBs for Rubik's cube. Also, we beat the fastest solver currently available by a factor of four.

The 4-peg Towers of Hanoi Problem

The classic Towers of Hanoi problem has three pegs, with a simple recursive optimal solution. For the 4-peg problem a recursive solution strategy has been proposed as well, but, absent a proof, search is the only way to verify the optimality of this solution (Frame 1941; Stewart 1941).

For the 4-peg Towers of Hanoi problem exponential memory algorithms detecting duplicates perform best. We use breadth-first heuristic search (BFHS) (Zhou and Hansen 2006). BFHS searches the problem space in breadth-first order but uses f-costs to prune states that exceed a cost threshold. We only need to perform one iteration with the presumed cost of an optimal solution as the threshold.

Lossy compression methods using cliques and their generalization are very effective for the 4-peg Towers of Hanoi problem (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, compress it to fit in memory, and use the remaining discs in a small uncompressed PDB. The state of the art for this problem (Korf and Felner 2007) uses a 22-disc PDB compressed to the size of a 15-disc PDB by ignoring the positions of the seven smallest discs. We limit our experiments to PDBs that can be constructed in two gigabytes of memory. Thus, our largest PDB uses 16 discs.

Experimental results using a 16-disc and a 2-disc PDB on the 18-disc problem with different levels of compression are

#	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 5: Solving the 18-disc Towers of Hanoi problem using a 16-disc PDB

shown in Table 5. The columns are the same as in Table 1. With a maximum entry of 161 and one byte per entry, the uncompressed 16-disc 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 and the fifth row ignore the smallest two and three discs, respectively. One can see that very little information is lost using lossy compression.

As mentioned earlier, the state of the art for this problem uses a PDB with as many discs as possible, compressed to fit in memory by ignoring a set of the smallest pattern discs. It is possible to compress these inconsistent PDBs even further storing the heuristic values modulo some number. The PDBs compressed by the positions of the smallest one and two disc have a maximum IRE of two and three, respectively. Therefore, both can be stored using three bits per entry. The PDBs compressed by the positions of the smallest three and four discs have a maximum IRE of five and seven, respectively. Thus, both can be stored using four bits per entry and be compressed by another factor of two, assuming one byte per entry suffices otherwise. Ignoring more than four discs requires more than four bits per entry.

In short, because of cliques and generalized cliques lossy compression is so powerful that we can only achieve a slight improvement with our compression methods. We can only compress by a factor of at most five, while Felner et al. can always compress by another factor of four by ignoring an additional disc with a very small impact on performance.

The Sliding-Tile Puzzles

The best existing heuristic for the 24 puzzle is the 6-6-6-6 partitioning and its reflection about the main diagonal. No major improvements using compression were reported by Felner et al. (2007). As mentioned earlier, Korf and Felner (2002) compressed these 6-tile PDBs by the position of the blank, making them inconsistent. While constructing these 6-tile PDBs, we calculated their maximum IRE, which is 7. Therefore, we could store the heuristic values modulo 15 using four bits per entry. However, the uncompressed PDBs can be stored in four bits per entry by storing just the addition above the Manhattan distance. Therefore, there is no advantage of using our compression method in this domain.

Conclusions

We have introduced a lossless compression method for PDBs which stores a consistent heuristic in just 1.6 or alternatively, two bits per state. For the Top-Spin puzzle and for Rubik's cube, 1.6 and two-bit PDBs are the best known compressed PDBs. For Rubik's cube, our best implementation beats 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 4-pegs Towers of Hanoi problem and the 24 puzzle we were able to report only minor or no improvements at all. In general, two-bit PDBs are useful where lossy compression using cliques or their generalization is not possible, or where adjacent entries in the PDB are not highly correlated.

Acknowledgment

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

References

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.

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. *Systems Science and Cybernetics* 4(2):100–107.

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.; 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.

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

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. 2006. Breadth-first heuristic search. *Artif. Intell.* 170(4):385–408.