Journal of Artificial Intelligence Research 30 (2007) 213-247 Submitted 12/06; published 10/07

Compressed Pattern Databases

Ariel Felner FELNER@BGU.AC.IL
Department of Information Systems Engineering,
Ben-Gurion University of the Negev
Beer-Sheva, Israel, 85104
Richard E. Korf KORF@CS.UCLA.EDU
Department of Computer Science
University of California Los Angeles
Los Angeles, CA, 90095
Ram Meshulam MESHULR1@CS.BIU.AC.IL
Department of Computer Science
Bar-Ilan University
Ramat-Gan, Israel, 52900
Robert Holte HOLTE@CS.UALBERTA.AC.CA
Department of Computing Science
University of Alberta
Edmonton, Canada

Abstract

A pattern database (PDB) is a heuristic function implemented as a lookup table that stores the lengths of optimal solutions for subproblem instances. Standard PDBs have a distinct entry in the table for each subproblem instance. In this paper we investigate compressing PDBs by merging several entries into one, thereby allowing the use of PDBs that exceed available memory in their uncompressed form. We introduce a number of methods for determining which entries to merge and discuss their relative merits. These vary from domain-independent approaches that allow any set of entries in the PDB to be merged, to more intelligent methods that take into account the structure of the problem. The choice of the best compression method is based on domain-dependent attributes. We present experimental results on a number of combinatorial problems, including the four-peg Towers of Hanoi problem, the sliding-tile puzzles, and the Top-Spin puzzle. For the Towers of Hanoi, we show that the search time can be reduced by up to three orders of magnitude by using compressed PDBs compared to uncompressed PDBs of the same size. More modest improvements were observed for the other domains.

1. Introduction and Overview

Heuristic search algorithms such as A* (Hart, Nilsson, & Raphael, 1968) and IDA* (Korf, 1985) find optimal solutions to state space search problems. They are guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of reaching node n from the initial state, and $h(n)$ is a heuristic function that estimates the cost of reaching a goal state from node

n. If $h(n)$ is "admissible", which means that it never overestimates actual cost, then these algorithms are guaranteed to find an optimal solution path if one exists.

Pattern databases are admissible heuristic functions implemented as lookup tables stored in memory (Culberson & Schaeffer, 1998). They are the best known heuristics for a number of combinatorial problems. In this paper we investigate the idea of compressing pattern database heuristics in order to improve the accuracy of such heuristics, for a given amount of memory.

We begin by describing the three different problem domains used in this paper, in order to ground all of our discussion in concrete examples.

1.1 Problem Domains

1.1.1 The 4-peg Towers of Hanoi

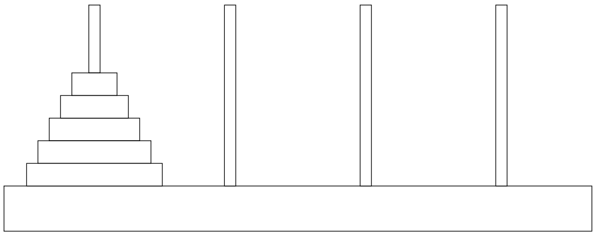

Figure 1: Five-disc four-peg Towers of Hanoi problem

The well-known three-peg Towers of Hanoi problem consists of three pegs and n discs of all different sizes which are initially stacked in decreasing order of size on one peg. The task is to transfer all the discs from the initial peg to a goal peg. Only the top disc on any peg can be moved, and a larger disc can never be placed on top of a smaller disc. For the three-peg problem, there is a simple recursive algorithm that provably returns an optimal solution. The idea is to move the $n-1$ smallest discs to the intermediate peg, then move the largest disc from the initial peg to the goal peg, and finally move the $n-1$ smallest discs from the intermediate peg to the goal peg.

The four-peg Towers of Hanoi problem (TOH4), (Hinz, 1997) shown in Figure 1, is more interesting. The recursive algorithm for the three-peg problem doesn't yield optimal solutions, because there are two intermediate pegs, and we don't know a priori how to distribute the $n-1$ smallest discs over these intermediate pegs in an optimal solution. There exists a deterministic algorithm for finding a solution, and a conjecture that it generates an optimal solution (Frame, 1941; Stewart, 1941), but the conjecture remains unproven (Dunkel, 1941). Thus, systematic search is currently the only method guaranteed to find optimal solutions for problems with a given number of discs.

1.1.2 The Sliding-Tile Puzzles

One of the classic domains in the AI literature is the sliding-tile puzzle. Three common versions of this puzzle are the 3x3 8-puzzle, the 4x4 15-puzzle and the 5x5 24-puzzle. Each consists of a square frame containing a set of numbered square tiles, and an empty position called the blank. The legal operators are to slide any tile that is horizontally or vertically

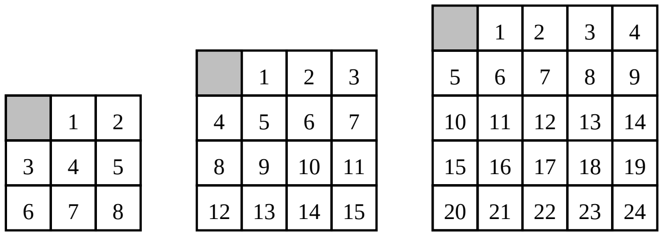

Figure 2: The 8-, 15- and 24-puzzle goal states

adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. The 8-puzzle contains 181,440 reachable states, the 15-puzzle contains over 10^{13} reachable states, and the 24-puzzle contains almost 10^{25} reachable states. The traditional goal states of these puzzles are shown in Figure 2.

1.1.3 The Top-Spin Puzzle

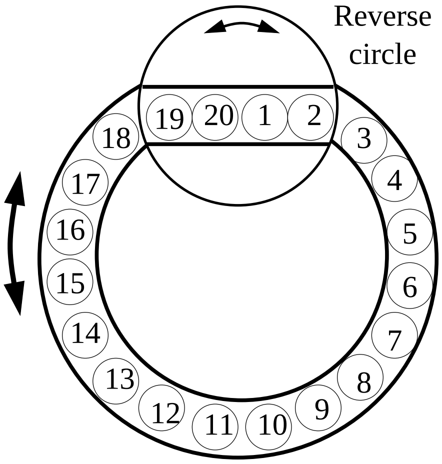

Figure 3: (20,4)-Top-Spin Puzzle

The (n,r)-Top-Spin puzzle has n tokens arranged in a ring. The ring of tokens can be shifted cyclically clockwise or counterclockwise. The tokens pass through the *reverse circle* which is fixed in the top of the ring. At any given time r tokens are located inside the reverse circle. These tokens can be reversed (rotated 180 degrees). The task is to rearrange the puzzle such that the tokens are sorted in increasing order. The (20,4) version of the puzzle is shown in figure 3 in its goal position where tokens 19, 20, 1 and 2 are in the reverse circle and can be reversed. Our encoding of this puzzle has N operators, one for each clockwise circular shift of length $0 \dots N-1$ of the entire ring followed by a reversal/rotation for the tokens in the reverse circle. Each operator has a cost of one. This is the same encoding analyzed by (Chen & Skiena, 1996). Practically, this puzzle can be implemented as a cyclic buffer where each operator reverses a set of r consecutive tokens[1]. Note that there are $n!$ different possible ways to permute the tokens. However, since the puzzle is cyclic, only the relative location of the different tokens matters, and thus there are only $(n-1)!$ different unique states.

1.2 Pattern Database Heuristics

Heuristics are typically implemented as functions from the states of the domain to a non-negative number. For example, a well-know heuristic for the sliding-tile puzzles is the *Manhattan distance*. It is computed by determining for each tile the minimum distance in grid units that it must travel to reach its goal location, and summing these values for all tiles except the blank. It is a lower bound on the optimal solution length, because each tile must move at least its Manhattan distance, and each move only moves one tile.

Pattern databases (PDBs) are heuristics in the form of lookup tables. A pattern database stores in memory the cost of an optimal solution to each instance of a subproblem of the original problem. These costs are then used as admissible heuristics for the original problem. PDBs have been used as lower bounds for combinatorial puzzles (Culberson & Schaeffer, 1998; Korf, 1997; Korf & Felner, 2002; Felner, Korf, & Hanan, 2004a; Felner, Meshulam, Holte, & Korf, 2004b), multiple sequence alignment (Zhou & Hansen, 2004; McNoughtton, Lu, Schaeffer, & Szafron, 2002; Schroedl, 2005; Edelkamp & Kissmann, 2007), vertex cover (Felner et al., 2004a), and planning problems (Edelkamp, 2001).

For example, consider a four-peg Towers of Hanoi problem with more than 15 discs. If we ignore all but 15 of the discs, there are $4^{15} = 2^{30}$ different possible configurations of these 15 discs. Now imagine an array which contains an entry for each possible configuration of these 15 discs, whose value is the exact number of moves required to optimally solve the corresponding 15-disc problem instance. Note that it doesn't matter which discs we choose, since their absolute sizes don't matter. As long as they all have different sizes only their relative size matters. If each entry will fit in a byte of memory, this table will occupy exactly one gigabyte of memory. This is an example of a PDB.

To build this PDB, we execute a complete breadth-first search starting with all 15 discs on the goal peg. As each configuration is encountered for the first time, we store its search depth in the corresponding entry in the PDB. A PDB of a given size is only built once, and can be reused to solve multiple problem instances with the same goal state.

We can use this PDB as a heuristic to solve any four-peg Towers of Hanoi problem with 15 or more discs. For each state encountered in the search, we look up the configuration of any given subset of 15 discs in the PDB, and use the stored value as a heuristic for the full state. Since this value is the number of moves needed to get the subset of 15 discs to the goal peg, it is a lower bound on the total number of moves needed to get all the discs in the problem to the goal peg.

1.3 Compressed Pattern Databases

The size of a PDB is the number of entries it contains. In general, the larger a PDB is, the more accurate it is, and the more efficient a search algorithm using that heuristic is. For

1. In the physical puzzle, we first have to rotate the ring such that these tokens will be in the reverse circle.

example, for solving a TOH4 problem with 20 or more discs, the values in a PDB based on 20 discs will be more accurate than those based on 15 discs. The drawback of large PDBs, however, is the amount of memory they consume.

The main idea of this paper is to compress PDBs that are too large to fit into memory in their uncompressed form into a size that will fit in memory. The compressing is done by partitioning the original PDB into groups of entries. Each group of entries in the original PDB is mapped to a single entry in the compressed PDB. In order to preserve admissibility, the value that is stored in the compressed PDB is the minimum among all the values in the group in the original PDB.

For example, given a PDB for the four-peg Towers of Hanoi problem based on 20 discs, we can divide the discs into the 15 largest discs and the 5 smallest discs. We then partition the PDB entries into 4^{15} groups of entries based on the positions of the 15 largest discs. Each group will have $4^5 = 1024$ different entries which correspond to the all the different positions of the 5 smallest discs. In the compressed PDB we only store one entry for each of these groups. Each entry in the compressed PDB will correspond to a different configuration of the 15 largest discs, and its value will be the minimum value over all configurations of the 5 smallest discs from the original PDB, in order to preserve admissibility.

Note that in general, the values in the compressed PDB will be much larger and hence more accurate than the corresponding values in a simple 15-disc PDB, despite the fact that the two databases are of the same size. The reason is that each entry of a simple 15-disc PDB is just the number of moves needed to solve the corresponding 15-disc problem, whereas each entry of the compressed PDB is the minimum number of moves needed to solve any instance of the 20-disc problem in which the 15 largest discs are in one particular configuration.

1.4 Overview

The primary questions we address in this paper are how to make the best use of a given amount of memory with compressed PDBs, and how to determine which entries of the PDB to compress. Specifically we make the following contributions:

- We introduce a number of methods for compressing PDBs. These methods vary from general methods to more constrained methods. The most general methods often result in a significant loss of information, while methods that rely on the structure of the underlying problem space are necessarily domain specific.

- We show that the best compression methods are often domain dependent, and provide guidelines for choosing the most promising method for a given domain. Experiments on the 4-peg Towers of Hanoi, the sliding-tile puzzles and the Top-Spin puzzle show that given the same amount of memory the search effort can many times be reduced by using compressed PDBs over using uncompressed PDB of the same size.

- We also describe methods for generating PDBs using external memory and compressing them to the size of the available memory.

This paper is organized as follows. We first provide definitions that will be used throughout the paper. We then consider how to build PDBs, and in particular, how to build compressed PDBs when the original PDB won't fit in memory. Then we discuss the different compressing methods. Next, we present our experimental results for the Towers of Hanoi, the sliding-tile puzzles and the Top-Spin Problem. Finally we offer our conclusions. A preliminary version of this paper appeared earlier (Felner et al., 2004b).

2. Definitions

We begin by providing an abstract characterization for search spaces of combinatorial problems, with sufficient structure to define PDBs and associated constructs. These definitions will be used throughout the paper.

2.1 Combinatorial Problems as Vectors of State Variables

A *problem space* is usually described abstractly as a set of atomic states, and a set of operators that map states to states. This corresponds to a labeled graph, called the problem-space graph. In addition, a specific problem instance is a problem space together with a particular initial state and a (set of) goal state(s). The task is to find an optimal path from the initial state to a goal state.

A *state* in a *combinatorial problem* can be described as a vector of *state variables*, each of which is assigned a particular *value*. For the domains studied in this paper, the variables correspond to the different *objects* of the problem, and the values correspond to the different *locations* that they can occupy. For example, in the Towers of Hanoi problem, there is a variable for each disc, with a value that indicates which peg the disc is on. For the sliding-tile puzzles, there is a variable for each physical tile and one for the blank, with a value that indicates the position occupied by that tile. For the Top-Spin puzzle, there is a variable for each token, whose value indicates its position. For convenience, in this paper, we will often refer to the variables as objects, and the values as locations, but in general these latter terms are simply convenient synonyms for variables and values, respectively. The *size* of a problem space is the number of distinct legal combinations of value assignments to the variables that are reachable from a given initial state.

A combinatorial problem is a *permutation problem* if the number of values equals the number of variables, and each value can only be assigned to one variable. For example, both Top-Spin and the sliding-tile puzzles are permutation problems because each location can only be occupied by one object. The Towers of Hanoi is not a permutation problem because each peg can hold more than one disc.

An *operator* in this formulation is a partial function from a state vector to a state vector. An operator changes the values of some of the variables. Note that the number of variables that have their values changed might be different for different domains and even for different operators in the same domain. For the Towers of Hanoi, the value of only one variable is changed by each operator, since only one disc moves at a time. For the sliding-tile puzzles two variables change values with each operator, because one physical tile and the blank exchange locations in each move. For Top-Spin, values of four variables are changed since four tokens change their locations.

The *goal* is a specified state or set of states. For permutation problems, such as the sliding-tile puzzles and Top-Spin, the goal state is usually a canonical state where object i

is in location i. The standard goal state for the Towers of Hanoi is where all the discs are on a single goal peg, and thus all the state variables have the same value.

2.2 Pattern Databases

Given a vector of state variables, and a set of operators, a subset of the variables defines a *subproblem* where we only assign values to variables in the subset, called *pattern variables*, while the values of the other remaining variables are treated as *don't cares*. For example, in the Towers of Hanoi, a subproblem would only include a subset of the discs.

A *pattern* is a specific assignment of values to the pattern variables. For example, a pattern might be a particular configuration of a subset of the discs.

A *pattern space* is the set of all the different reachable patterns of a given subproblem. For example, the pattern space of a four-peg Towers of Hanoi subproblem with P pattern variables includes all the different possible assignments of the P discs to the pegs, and is of size 4^P. For permutation problems of n elements with P pattern variables, the pattern space typically has $n \times (n-1) \times \ldots \times (n-P+1)$ states.

Each state in the original state space is *projected* onto a pattern of the pattern space by only considering the pattern variables, and ignoring the other variables. We refer to the set of states that project to the same pattern as the *states* of that pattern.

A *goal pattern* is the projection of a goal state onto the pattern variables. Multiple goal states may give rise to multiple goal patterns, or just a single goal pattern.

There is an *edge* between two different patterns p_1 and p_2 in the pattern space if and only if there exist two states s_1 and s_2 of the original problem, such that p_1 is the projection of s_1, p_2 is the projection of s_2, and there is an operator of the original problem space that connects s_1 to s_2.[2] The effect of the operator on the patterns is called a *pattern move*.

The *distance* between two patterns in the pattern space is the number of edges or pattern moves on a shortest path between the two patterns.

The distance between two patterns p_1 and p_2 in the pattern space is therefore a lower bound on the shortest distance between any pair of states s_1 and s_2 such that p_1 is the projection of s_1 and p_2 is the projection of s_2.

A *pattern database* (PDB) is a lookup table that includes an entry for each pattern of the pattern space. The value stored for a pattern is the distance of the pattern in the pattern space from any goal pattern. A PDB value stored for a given pattern is therefore an admissible heuristic for all the states that project onto that pattern.

3. Building Pattern Databases

In general, a PDB is built by running a breadth-first search in the pattern space backwards from the goal pattern until the entire pattern space is spanned. However, since the operators of the problem apply to states of the original problem, and not to the patterns directly, building the PDB is slightly different for the different domains.

For the Towers of Hanoi problem, we simply ignore all non-pattern discs. In other words, if there are P discs in the pattern, we simply search a Towers of Hanoi problem space with P discs. For each state, we keep track of its depth in the breadth-first search.

2. This is sometimes called *edge homomorphism*.

For the sliding-tile puzzles, we have to keep track of the blank position, even if it is not included in the pattern, in order to determine whether a move is legal or not. Thus, a state in this search is uniquely determined by the positions of the pattern tiles and the blank. This is equivalent to a problem space that includes all the tiles, but in which the non-pattern non-blank tiles are indistinguishable from each other.

In the original application of PDBs to the sliding-tile puzzles (Culberson & Schaeffer, 1998), all moves were counted in the PDB values. The drawback of this approach is that with multiple PDBs, the only way to combine their values without sacrificing admissibility is to take their maximum value. Additive pattern databases (Korf & Felner, 2002) allow us to sum the values of multiple pattern databases without violating admissibility, and are much more effective when they are applicable. In order to construct additive pattern databases for the sliding-tile puzzles, we can only count moves of the pattern tiles, and ignore moves of non-pattern tiles. We adopt this approach here.

For Top-Spin, the goal is to put the tokens in their correct cyclic order, but our representation is linear rather than cyclic. To reconcile these two different representations, we always keep token one in location one. If an operator moves token one, this is immediately followed by a cyclic shift of all the tokens to restore token one to location one, at no additional cost. If our pattern doesn't include token one, then we must keep track of the position of token one, in order to correctly implement these shifts. With each state, we store only the number of moves that involve at least one pattern token, either in the reversal, or in the subsequent shift to restore token one to position one. For example, if a reversal changes the position of token one, it will be counted as a pattern move even if the reversal doesn't include any pattern tokens, because the pattern tokens will move in the subsequent cyclic shift of all the tokens.

When each pattern is first generated, the number of pattern moves needed to reach that state is stored in the corresponding entry in the PDB. The PDB only needs to be built once for a specific goal state.

3.1 Mapping Patterns to PDB Indices

PDBs are sometimes implemented by sophisticated data structures such as lexicographic trees (Felner et al., 2004a) or octrees (McNoughtton et al., 2002). In addition, *symbolic pattern databases* (Edelkamp, 2002) are based on binary decision diagrams (BDDs) (Dunkel, 1992). Nevertheless, PDBs are most commonly implemented by arrays of entries, each storing a heuristic value for a specific pattern. Thus, for simplicity, in this paper we assume that PDBs are implemented by arrays.

To save memory, we want to avoid having to store the patterns themselves along with their heuristic values. This is done by representing each pattern with a unique index in the PDB. The particular mapping from patterns to indices is domain specific.

For example, a state of the Towers of Hanoi problem can be uniquely represented by specifying the peg that each disc is on, since all the discs on any peg must be in sorted order by size. For the four-peg problem, a peg can be specified by two bits, and a complete problem state can be uniquely specified by a string of $2n$ bits, where n is the number of discs. Furthermore, every such bit string represents a legal state of the problem.

For permutation problems, there are two obvious ways to store a PDB with k variables - a *sparse mapping* and a *compact mapping*.

- **sparse mapping -** The simplest organization is a k-dimensional array, with each dimension having a range of 0 to $n-1$. An individual pattern is mapped into the table by taking the value of each pattern variable as a separate index into the array. For example, if we have a pattern of three variables (X,Y,Z), whose values are (2,1,3) respectively, it would be mapped to the array element $A[2][1][3]$. The total size of such an array for all configurations of k elements is n^k. This is called a *sparse mapping*. The advantage of sparse mapping is that it is simple to implement and the indices can be efficiently computed. The disadvantage is that it is not efficient in terms of space, since it wastes those entries with two or more equal indices, since such entries do not correspond to valid configuration.

- **compact mapping -** Another method for indexing PDBs for permutation problems is called a *compact mapping*. In particular, each permutation of k elements into n possible locations can be mapped bijectively to a unique index in the range 0 to $n \times n-1 \times \ldots \times n-k+1$. One such mapping maps each permutation to its index in a lexicographic ordering of all such permutations. For example, assuming $k = n = 3$, the permutation (2 1 3) is mapped to the third index, since it is preceded by (1 2 3) and (1 3 2) in lexicographic order. The advantage of compact mapping is that it does not waste any space. The disadvantage is that computing the indices is more complex. (Myrvold & Ruskey, 2001) provide linear-time algorithms for bijective mappings between permutations and integers, and (Korf & Shultze, 2005) provide linear-time algorithms in which the index of a permutation represents its position in lexicographic order. At least in our experiments (reported below), even with this advanced mapping algorithm, the access time of the sparse mapping was faster than its compact mapping counterpart.

It is worth noting that the memory savings of the compact mapping over the sparse mapping decreases as the number of variables in the domain increases beyond the number of elements in the PDB. For example, consider a 6-tile PDB for the 24 Puzzle. Sparse mapping requires $25^6 = 244 \times 10^6$ entries, while compact mapping uses only $25 \times 24 \times \ldots \times 20 = 128 \times 10^6$ entries. Indeed, since the memory savings are modest, and sparse mapping is simpler it has often been used to implement the 6-tile PDBs (Korf & Felner, 2002; Felner et al., 2004a; Felner, Zahavi, Holte, & Schaeffer, 2005; Zahavi, Felner, Holte, & Schaeffer, 2006). Similarly, sparse mapping was also used to implement the 5-tile PDBs of the 35 puzzle (Felner et al., 2004a).

3.2 Building Large Pattern Databases

Compressed PDBs are used when there isn't sufficient memory to store the uncompressed PDBs. This raises the question of how to generate a compressed PDB without exhausting memory in the first place.

Consider a PDB for the four-peg Towers of Hanoi problem for example. Since a state can be uniquely represented by a bit string index of length $2n$, only the heuristic values are stored. If we use an unsigned character array, we can store values up to 255 in one byte, which is sufficient for the maximum number of moves needed to solve any problem with up to 18-discs (Korf, 2003). Thus, a 15-disc PDB, or a PDB compressed to the size of a 15-disc PDB, would occupy 4^{15} bytes, or a gigabyte of memory. A 16-disc PDB would need four gigabytes, however. Given a machine with two gigabytes of memory, for example, how can we generate a 16-disc PDB compressed to the size of a 15-disc PDB?

To generate such a PDB, we must perform a complete breadth-first search of the 16-disc problem space, starting from the standard goal state. As each state is generated, the largest 15 discs are used as an index into the PDB, and if the entry is empty, the search depth is stored. A breadth-first search is normally implemented by a first-in first-out queue of nodes that have been generated, but not yet expanded. Initially, the goal state is placed in the queue, and at each step we remove and expand the node at the head of the queue, and append its children to the tail of the queue. We can keep track of the search depth by placing a marker in the queue between nodes of successive depths. The maximum size of this queue is determined by the maximum number of nodes at any depth of the search. For example, for the 16-disc problem, this number is $162,989,898$ nodes at depth 134 (Korf, 2003). Since a state of the 16-disc problem can be stored in 32 bits, this queue requires $651,959,592$ bytes, or 622 megabytes of memory.

In order for this breadth-first search to terminate, however, we have to be able to detect whether or not we have encountered a particular state before. An efficient way to do this is to store a bit array, with one bit for each state, initialized to all zeros. Whenever a state is first encountered, its corresponding bit is set to one. Whenever we generate a node, we check its bit, and discard the node if its bit is already set to one. For the 16-disc problem we need 4^{16} bits, or 512 megabytes for this array.

622 megabytes for the queue, plus 512 megabytes for the bit array, and one gigabyte for the pattern database exceeds the capacity of a two gigabyte machine. We can't use the PDB to replace the bit array, because each PDB entry represents four different states of the 16-disc problem, each of which has to be separately detectable.

In the case of compressing a 16-disc PDB, the solution is simple. Since the breadth-first search queue is FIFO, all of its accesses are sequential, and it can be efficiently stored on magnetic disk instead of memory. A simple implementation is to keep two different files, one for nodes at the current depth, and the other for nodes at the next depth. The first file is read sequentially, and children are appended to the tail of the second file. When the first file is exhausted at the end of the current search depth, the second file becomes the new first file, and another file is created for the children at the next depth. Moving the queue to disk creates enough space to store both the bit array and the PDB in memory.

Unfortunately, this won't work for compressing larger PDBs. For example, a complete breadth-first search of the 17-disc problem will require all two megabytes just for the bit array. The bit array cannot be stored on disk, because it is accessed randomly, and random access of a byte on disk requires an average of about 5 milliseconds latency.

Techniques such as *delayed duplicate detection* have been developed for performing large breadth-first searches on disk, without storing a bit for each state in memory (Munagala & Ranade, 1999; Korf, 2003, 2004; Korf & Shultze, 2005). The key idea is to not immediately check each node to see if it has previously been generated, but delay the duplicate detection until large numbers of duplicates can be eliminated by sequentially accessing the nodes on disk. Using such techniques, complete breadth-first searches have been performed on the four-peg Towers of Hanoi problem with up to 22 discs (Korf & Felner, 2007).

Breadth-first search with delayed duplicate detection can be performed using relatively little memory, allowing a compressed PDB to be built in memory at the same time. However, the technique is more efficient with more memory. Thus, as an alternative, we can build the compressed PDB in two phases.

The first phase performs the breadth-first search on disk, without the compressed PDB in memory. During this phase, as each state is expanded, we write out to another disk file an ordered pair consisting of the index of the state in the compressed PDB, followed by its depth in the breadth-first search. This file is simply a linear list of these pairs, and is written sequentially. Note that the depths in this file will be in sorted order.

In the second phase, we build the compressed PDB in memory, by sequentially reading this file. For each ordered pair, we look up the index in the PDB. If it is empty, we store the corresponding depth in the PDB, and if it is full, we simply ignore the ordered pair. Finally, we write the compressed PDB to disk for use in future searches, and delete the file of ordered pairs. This scheme allows us to use almost all of our memory for the breadth-first search, regardless of the size of the compressed PDB.

4. Compressing Pattern Databases

In most previous studies, PDBs have had one entry for each pattern in the pattern space. In this section, we describe different methods for compressing PDBs by merging a number of PDB entries into a single entry. In order to preserve admissibility, the merged entry will store the minimum value of the entries merged, with a consequent *loss of information* for those original entries with larger values than the minimum.

The idea to compress large tables into a smaller size by merging several entries together has been suggested in the past outside of the AI community, e.g., in the form of bit-state hashing (Bloom, 1970) and in the form of hash compaction (Stern & Dill., 1995).

The ideal compression scheme would group together all entries that have the same value. This would preserve all the information, and result in a PDB whose size is the number of distinct values in the original PDB. Of course, this is almost impossible to achieve in practice.

To compress a PDB by a factor of k, we typically partition the entire pattern space into M/k sets of k patterns each, where M is the size of the original PDB. The compressed PDB will contain one entry for each set. The key to making this method effective is that the values stored in the original PDB for the k entries should be as close to one another as possible to minimize the resulting *loss of information*. One of the questions addressed in this paper is how to identify PDB entries with similar values to be merged.

4.1 Compressing Nearby Patterns

An effective heuristic to identify PDB entries with similar values is to compress entries that correspond to patterns that are close together in the pattern space. Assume that two patterns p_1 and p_2 are close to each other in the pattern space, meaning that the distance between them is small. Furthermore, assume that all operators are reversible. Then the distances from p_1 to the goal pattern and from p_2 to the goal pattern (i.e., their PDB values) are also similar. More formally:

$$(d(p_1, p_2) = c) \Longrightarrow (|PDB(p_1) - PDB(p_2)| \leq c).$$

If these two patterns are compressed into the same entry we are guaranteed that the loss of information is at most c moves. Therefore, we would like to compress the PDB such that patterns that are mapped to the same entry are close to each other in the pattern space.

We provide a number of different methods for compressing PDBs. The most general methods often result in significant loss of information, while methods that rely on the structure of the underlying problem space are necessarily domain-specific.

4.2 Compression Based on General Mapping Functions

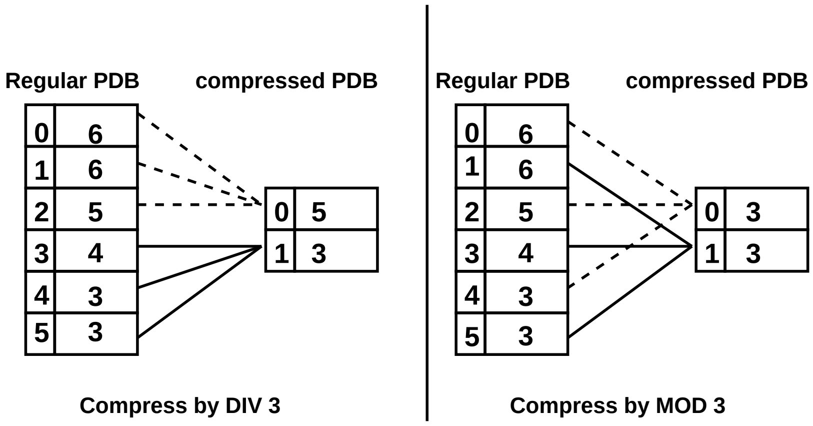

Figure 4: Compressed pattern database

The easiest way to compress a PDB is to use a general function that maps exactly k original patterns to each compressed entry. This compresses a PDB with M entries to a compressed PDB with M/k entries. Two examples of such functions for mapping indices i in the original PDB to the compressed PDB are:

- $Compressed\_index(i) = i \;\; DIV \;\; k$
- $Compressed\_index(i) = i \;\; MOD \;\; k$

where *DIV* is integer division by k, and *MOD* is the remainder after division by k. The advantage of these mapping functions is that they are general, and thus a PDB of size M can be compressed to fit into any amount of available memory.

Figure 4 shows these two ways to compress a PDB of size 6 by a factor of 3 into a compressed PDB of size 2. Note that the values of the specific PDB of the figure are locally correlated with their indices. Therefore, it is easy to see that in this case the *DIV* operator is better than the *MOD* operator since it compresses more highly correlated values and therefore the resulting the loss of information is smaller.

The challenge is to build the PDB such that entries with similar indices come from patterns that are close to each other in the pattern space. Such values are locally correlated.

Given locally correlated values, it is better to compress them with the *DIV* operator, rather than compressing an arbitrary set of k entries. As will be shown below, for many domains it is possible to build a PDB such that locally correlated entries correspond to nearby patterns.

4.3 Compression Based on Individual Variables

To compute any PDB, we use a function to map each state to an index in the PDB, which uniquely represents the corresponding pattern. Many of these functions map the values of particular variables to particular bits of the index. For example, as described above, the most natural representation of a state of the n-disc four-peg Towers of Hanoi problem is a bit string of $2n$ bits. In this representation, pairs of bits represent the locations of different discs. The same is true of the sparse representation described above for permutation problems, where different indices of the multi-dimensional array encode the locations of different tiles or tokens. The situation is a little more complex for the compact representation of permutation problems, but the same principle applies.

To compress these PDBs, we can map the indices in the original PDB to indices in the compressed PDB by ignoring certain bits. If these bits correspond to particular variables, then this has the effect of compressing entries by ignoring the values of those variables. If the operators of the problem space only change the values of a small number of variables, then patterns that differ in only a few variables will tend to be close together in the pattern space, and their distances to the goal will also tend to be similar. Thus, when compressing these entries the loss of information should be small.

In both the compact and sparse mappings described above we calculate the index in the PDB according to a predefined order of the variables. Assume that the last variable in that order has q different possible values. For the sparse mapping of permutation problems $q = n$, while for the compact mapping $q = n - P + 1$, where n is the total number of variables, and P is the number of pattern variables. If we divide the index by q, then we are compressing entries that differ only in the value of the last variable. In our problems this means that in a set of entries that are compressed together, all the pattern objects but the last one are located in the same location. For example, all patterns which are indexed as $PDB[a][b][c][d]$ will be compressed to $COMPRESSED\_PDB[a][b][c]$. The disadvantage here is that we are forced to compress by a factor of n in the sparse mapping case, or $n-P+1$ in the compact mapping case, even if this doesn't exactly fit the available memory.

The idea of compressing the last variable can be generalized to compressing the last two or more variables. The compression factor would be n, n^2, n^3 etc. for the sparse mapping, and $n - P + 1$, $(n - P + 1) \cdot (n - P + 2)$, $(n - P + 1) \cdot (n - P + 2) \cdot (n - P + 3)$ etc. for the compact mapping. Thus, while this method may achieve high correlation in the values of compressed entries, it only allows compression by certain values.

4.3.1 Comparing Compressed and Uncompress PDBs of the Same Size

We denote an uncompressed PDB of a set of P variables as PDB_P, and similarly an uncompressed PDB of $P - C$ variables as PDB_{P-C}. We denote a PDB of P variables compressed by C variables $CPDB_{P/C}$. We now provide the following two propositions comparing PDB_{P-C} and $CPDB_{P/C}$.

1. The size of $CPDB_{P/C}$ is equal to the size of PDB_{P-C}.

Proof: It is easy to see that they are both of size $n^{|P-C|}$ using the sparse mapping, and $(n \times (n-1) \times (n-2) \times \ldots \times (n - |P| + |C| + 1))$ using the compact mapping.

2. Assume that P is a set of variables and that $C \subset P$. For each state s of the search space $CPDB_{P/C}(s) \geq PDB_{P-C}(s)$.

   Proof: Both mappings are based on the same $P - C$ variables. However, while PDB_{P-C} completely ignores the C variables, $CPDB_{P/C}$ contains the minimum values of PDB_P over all combinations of the C variables.

Thus, for a given memory size M, a PDB larger than M compressed by individual variables to the size of M is at least as accurate and usually more accurate than an uncompressed PDB of size M for the same variables.

4.4 Compressing Cliques

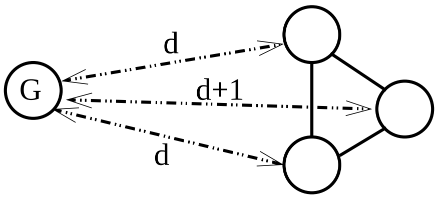

Figure 5: Cliques in PDBs

Suppose that a given set of patterns form a clique in the pattern space. This means that all the patterns in the set are reachable from each other by one move in the pattern space. Thus, the PDB entries for these nodes will differ from one another by no more than one, assuming that all operators are reversible. Some will have a value d, and the rest will have value $d + 1$ as shown in Figure 5. It is worthwhile to compress such cliques in the pattern space since the loss of information is guaranteed to be at most one move. If we can identify a general structure of q entries in the PDB that represent a clique of size q, we can map all these q patterns to one entry.

The existence of cliques in the pattern space is domain dependent. Furthermore, to take advantage of them, we need a compression function that will map members of a clique to the same index in the compressed PDB. These conditions will not exist in all problem spaces, but do exist in some of the domains we considered.

In combinatorial problems where the operators move only one object at a time, such as the Towers of Hanoi and the standard sliding-tile puzzles, cliques often represent states that differ only in the location of a single object. Therefore, compression based on that object (or variable) amounts to compressing cliques in practice. This coincidence does not necessarily occur in general. As will be shown below, for Top-Spin, patterns that only differ in the location of one object are not cliques, and for this domain the two compression methods do not coincide.

Compressing cliques of size q can be done in the following two ways:

- **lossy compression :** Store the minimum value of the q entries. The admissibility of the heuristic is preserved and the loss of information is at most one move.

- **lossless compression:** Store the minimum value for the q entries. Assume this value is d. Store also q additional bits, one for each entry in the clique, that indicates whether the entry's value is d or $d+1$. This preserves all the information of the original PDB, but will usually require less memory.

The idea of compressing cliques can be generalized to a set of z nodes with a diameter of r. In other words, each pair of nodes within the set have at least one connecting path consisting of r or fewer edges. A clique is the special case where $r = 1$. We can compress this set of nodes into one entry by taking the minimum of their entries and lose at most r moves. Alternatively, for lossless compression we need an additional $z \cdot \lceil log(r+1) \rceil$ bits to indicate the exact value. If the size of an entry is b bits then it will be beneficial in terms of memory to use lossless compression for sets of nodes with a diameter of r as long as $\lceil log(r+1) \rceil < b$.

4.5 Inconsistency of the Compressed Heuristic

An admissible heuristic h is *consistent* if for any two states, x and y, $|h(x)-h(y)| \leq dist(x,y)$ where $dist(x,y)$ is the shortest path between x and y in the problem space. In particular, for neighboring states the h values of the two states differ by at most one move. An admissible heuristic h is *inconsistent* if for at least one pair of nodes x and y, $|h(x)-h(y)| > dist(x,y)$. Compressed PDBs can be inconsistent since the loss of information can be different for different states. For example, let x and y be neighboring states (with an edge cost of 1) such that they are respectively mapped to the patterns of lines 2 and 3 from the left frame of figure 4. While their original heuristics are consistent (5 and 4 respectively) their compressed heuristics are inconsistent (5 and 3). Pathmax (PMX) is one approach to correcting inconsistent heuristics (Mero, 1984). It propagates heuristic values from a parent node p to its child c as follows. $h(p) - dist(p,c)$ is a lower bound on $dist(c, Goal)$ and therefore can be used instead of $h(c)$ if it is larger. A new approach for further handling inconsistent heuristics is called *bidirectional pathmax* (BPMX) (Felner et al., 2005; Zahavi, Felner, Schaeffer, & Sturtevant, 2007). It generalizes PMX such that with BPMX, heuristic values are propagated in both directions in order to increase heuristic values of nodes in the neighborhood. Preliminary results of applying BPMX and compressed PDBs show that in some cases a small reduction in the number of generated nodes can be achieved.

We now present experimental results obtained with compressed pattern databases on each of our example domains.

5. The 4-peg Towers of Hanoi Problem (TOH4)

The state space of TOH4 has many small cycles, meaning there are many paths between the same pair of states. For example, if we move the same disc twice in a row, we can achieve the same effect with only a single move of the disc. As another example, if we move a disc from peg A to peg B, and then another disc from peg C to peg D, applying these two moves in the opposite order will have the same effect. If we forbid moving the same disc twice in a row, and only apply commutative operators in one order, we get a branching factor of 3.766. For the 7-disc problem, the optimal solution depth is 25 moves, and a complete search to this depth will generate 3.766^{25} nodes. However, there are only $4^7 = 16,384$ unique states.

Thus, any depth-first search, such as IDA*, will generate enormous numbers of duplicate nodes, and will be hopelessly inefficient in this domain.

Thus, in order to search in this domain we used Frontier-A* (FA*), a modification of A* designed to save memory (Korf, Zhang, Thayer, & Hohwald, 2005). FA* saves only 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, FA* stores those operators that lead to Closed nodes, and when expanding a node those operators are not used.

5.1 Additive Pattern Databases for TOH4

The applicability of PDB heuristics to TOH4 was first shown by (Felner et al., 2004a). Consider a 16-disc problem. We can build a PDB for the ten largest discs by including an entry for each of the 4^{10} legal patterns of these discs. The value of each entry is the minimum number of moves required to move ten discs from their corresponding positions to the goal peg, assuming there are no other discs in the problem. During the problem-solving search, given a state of the 16-disc problem, we compute the index corresponding to the ten largest discs, and look up the value for this configuration in the PDB. This value is an admissible heuristic for the complete 16-disc problem, because a solution to the 16-disc problem must move the largest ten discs to the goal peg, in addition to the smallest six discs

A similar PDB can be built for the six smallest discs. Values from the ten-disc PDB and the six-disc PDB can be added together to get an admissible heuristic value for the complete state. The reason is that a complete solution must move all the discs to the goal peg. Furthermore, each move only moves one disc, and the PDB values for these two subproblems only count moves of their respective pattern discs. Therefore, the sum of the PDB values from two disjoint sets of discs is a lower bound on the number of moves needed to solve the original problem. The idea of additive PDBs was first introduced by (Korf & Felner, 2002) for the sliding-tile puzzles. A deeper analysis of additive PDBs is provided by(Felner et al., 2004a).

Note that a PDB based on n discs will contain exactly the same values for the largest n discs, the smallest n discs, or any other set of n discs. The reason is that all that matters is that the discs be of different sizes, and not their absolute sizes. Furthermore, a PDB for m discs also contains a PDB for n discs, if $n < m$. To look up a pattern of n discs, we simply assign the $m - n$ remaining discs to the goal peg, and then look up the resulting pattern in the m-disc PDB. Thus, in practice we only need a single PDB for the largest number of discs of any group of our partition. In our case, a ten-disc PDB contains both a PDB for the largest ten discs and a PDB for the smallest six discs. In general, the most effective heuristic is based on partitioning the discs into groups that maximize the size of the largest group, subject to the available memory. The largest PDB we can use on a machine with a gigabyte of memory is for 14 discs. This has 4^{14} entries, and at one byte per entry occupies 256 megabytes. The rest of the memory is used for the Open list of FA*.

Given a PDB of 14 discs, there are two ways to use it for a 16-disc problem. The first is called *static partitioning*. In this method, we statically partition the discs into one group of 14 discs and the remaining 2 discs, and use the same partition for all the nodes of the

Heuristic	Path	Avg h	Nodes	Seconds
Static 13-3	161	72.17	134,653,232	48.75
Static 14-2	161	87.04	36,479,151	14.34
Dynamic 14-2	161	93.46	12,827,732	21.56

Table 1: Static vs Dynamic Partitioning for the 16-disc problem

search. The other method is called *dynamic partitioning*. For each state of the search, we compute all $16 \cdot 15/2 = 120$ different ways of dividing the discs into groups of 14 and 2, look up the PDB values for each pair, sum the two values, and return the maximum of these as the overall heuristic value. Here, the exact partitioning into disjoint sets of discs is dynamically determined for each state of the search. Table 1, taken from (Felner et al., 2004a), compares static partitioning and dynamic partitioning for solving the standard initial state of the 16-disc TOH4. Each row corresponds to a different PDB setting. The static partitions divide the discs into a large group of the largest discs and a smaller group of the smallest discs. The columns provide the optimal path length, the average heuristic value over the PDB, the number of generated nodes and amount of time taken to solve the standard initial state. A 14-2 split is much better than a 13-3 split since the 14-disc PDB is much more informed than the 13-disc PDB. For the 14-2 split, a dynamically partitioned heuristic is more accurate, the search generates fewer nodes, and therefore FA* requires less memory, but takes longer to run due to the multiple heuristic calculations for each state. Static partitioning is simpler to implement, and consumes much less time per node, but generates more nodes than dynamic partitioning, thus occupying more memory. Static and dynamic partitioning apply in any domain where additive PDBs apply.

5.2 Compressed Pattern Databases for TOH4

As explained above, states for TOH4 can be represented by bit strings, where pairs of adjacent bits represent the positions of particular discs. Therefore, compressing these PDBs is very easy. For example, if the smallest disc is represented by the least significant two bits, compression based on the smallest disc can be accomplished by a right shift of two bits in the bit string. Note that this is logically equivalent to *DIV 4*. We will refer to this as "compressing the smallest disc".

In practice, compressing the smallest disc amounts to compressing cliques in this domain. For TOH4, the largest cliques are of size four, since the smallest disc can always move among the four pegs without moving any of the larger discs. Thus, we can store a PDB of P discs in a table of size 4^{P-1}, instead of 4^P, by compressing the four states of the smallest disc into one entry.

In the compressed PDB, we will have one entry for each configuration of the $P-1$ largest discs. Lossy compression would store the minimum value of the four entries of the original PDB that correspond to each entry of the compressed PDB, and lose at most one move for some of the entries. Alternatively, lossless compression would store four additional bits in each entry of the compressed PDB, to indicate for each location of the smallest disc whether the value is the minimum value or one greater.

Heuristic	h(s)	Avg h	Nodes	Time	Mem
Static 14+2	116	87.04	36,479,151	14.34	256M
Static 13+3	102	72.17	134,653,232	54.02	64M
Static 12+4	90	59.01	375,244,455	184.62	16M
Static 11+5	74	47.32	$>$ 462,093,281	$>$ 243.15	4M

Table 2: Solving 16 discs without compression

This can be generalized to compressing the smallest two (or more) discs. We fix the position of largest $P-2$ discs, and consider the 16 different configurations of the two smallest discs. These form a set of nodes of diameter three. Thus, we can compress these 16 entries into one entry, and lose at most three moves for any state. Alternatively, for lossless compression we can add $2 \cdot 16 = 32$ bits to the one byte entry in the compressed PDB, for a total of five bytes, and store the exact values, compared to 16 bytes for the uncompressed PDB.

5.3 Experiments on the 16-disc 4-peg Towers of Hanoi

We now provide experimental results for the various compressing methods on the 16-disc 4-peg Towers of Hanoi problem. All the results are for solving the standard version of the problem, which is moving all discs from one peg to another. The optimal solution to this problem is 161 moves long. Unless stated otherwise, all the experiments in this paper were conducted on a 2.4Ghz PC with one gigabyte of main memory.

5.3.1 Uncompressed PDBs of Different Sizes

For comparison purposes, Table 2 shows results for uncompressed PDBs built by static partitioning of $14+2$, $13+3$, $12+4$ and $11+5$. The first column shows the heuristic that was used. In each case, the larger group contained the largest discs. The second column shows the heuristic value of the initial state. The next column is the average heuristic value over all entries of the larger PDB (the 14-disc PDB, the 13-disc PDB etc). The next columns present the number of generated nodes, the amount of time in seconds, and the amount of memory needed in megabytes for the larger PDB, to optimally solve the 16-disc problem. We weren't able to solve the problem with the $11+5$ heuristic before running out of memory for the Open list.

5.3.2 Compressing the Largest Discs

As explained above, in this domain compressing cliques is identical to compressing the smallest disc. A complementary method is to compress the largest discs. In our implementation the smallest discs were represented by the least significant bits and the larger discs were represents by the most significant bits. In this particular representation, compressing the largest z discs can be accomplished by masking off the corresponding $2z$ bits. This is logically equivalent to taking the representation $MOD\, 4z$, if the largest discs are represented by the most significant bits.

Discs	h(s)	Avg h	Nodes	Time	Mem
0	116	87.04	36,479,151	14.34	256M
.5	101	80.55	70,433,127	28.69	128M
1	100	72.17	285,190,821	143.78	64M
1.5	85	66.46	410,850,034	269.55	32M
2	84	59.01	791,374,842	543.10	16M
2.5	69	53.94	1,086,889,788	776.22	4M

Table 3: Solving 16 discs with a 14-2 PDB where the 14-disc PDB was compressed based on the large discs

Table 3 presents results for solving the 16-disc TOH4 problem by compressing the largest discs. We statically divided the discs into two groups. The largest fourteen discs define the 14-disc PDB. The smallest two are in their own group and have a separate PDB with $4^2 = 16$ entries. To compute the heuristic for a state, the values from the 2-disc PDB and the 14-disc PDB are added. The different rows of the table correspond to different degrees of compression of the 14-disc PDB. The 2-disc PDB only occupies 16 entries and hence there is no need to compress it.

The first column shows the number of largest discs that were compressed. The first row represents no compression. The whole numbered rows are based on masking off both bits used to represent a disc. The fractional numbers are based on masking off one of the two bits used to represent a disc. For example, the row with 1.5 in the first column is based on masking off both bits of the largest disc, and the most significant bit of the two bits for the next largest disc. The rest of the columns are in the same format as Table 2.

The table clearly shows that the running time increases significantly when compressing the largest discs. The reason is that the locations of the largest discs have a large impact on the PDB values. Thus, the values that are compressed here are not correlated with each other, and a great deal of information is lost. This provides evidence for our claim that values that are uncorrelated should not be compressed.

Comparing Table 3 to Table 2 shows that compressing the largest discs performs worse than a PDB of similar size without compression. For example, with 64 megabytes of memory it is better to solve the problem with a simple PDB of 13 discs plus a small PDB of 3 discs (2nd line of Table 2), than with a 14-disc PDB compressed by the largest disc, plus a small PDB of 2 discs (3rd line of Table 3). Thus, compressing the largest discs is not beneficial for TOH4.[3]

Discs	h(s)	Avg h	r	Nodes	Time	Mem
0	116	87.04	0	36,479,151	14.34	256M
1	115	86.48	1	37,964,227	14.69	64M
2	113	85.67	3	40,055,436	15.41	16M
3	111	84.45	5	44,996,743	16.94	4M
4	110	82.74	9	45,808,328	17.36	1M
5	103	80.85	13	61,132,726	23.78	256K
6	99	78.54	17	76,121,867	33.72	64K
7	98	74.81	25	97,260,058	36.63	16K
8	96	68.34	33	164,292,964	67.59	4K
9	75	62.71	41	315,930,865	155.22	1K
1 lls	116	87.04	0	36,479,151	15.87	96M

Table 4: Solving 16 discs where the 14-disc PDB was compressed by the small discs

5.3.3 Compressing Cliques or the Smallest Discs

Table 4 presents results for a 14-2 static partitioning of the 16 discs, but compressing the 14-disc PDB by the smallest discs. In our representation compression based on the smallest z discs can be accomplished by masking the left $2z$ bits. This is logically equivalent to performing *MOD* by $4z$. The different rows of the table correspond to compressing the 14-disc PDB by different numbers of discs. All the rows represent lossy compression, except for the last row which represent lossless compression (denoted as *lls*). The first row of Table 4 is for the complete 14-disc PDB with no compression. The second row corresponds to compressing the smallest of the 14 largest discs. In that case, the 14-disc PDB only contains 4^{13} entries which correspond to the different possible configurations of the 13 largest discs. For each of these entries, we store the minimum of the four possibilities for the smallest disc. This row corresponds to compressing cliques since all four possible positions of the smallest disc can be reached from one another with a single move. The third row compresses the two smallest discs of the 14 largest by storing the minimum of the 16 possibilities in a single entry and so on.

The most important result here is that when compressing the PDB by several orders of magnitude, most of the information is preserved. For example, compressing the five smallest of the 14 largest discs reduces the memory by factor of $4^5 = 1024$, but only increased the search effort by less than a factor of two in both the number of generated nodes and the time to solve the problem. When the PDB was compressed by a factor of $4^9 = 262,144$, the search effort only increased by a about a factor of ten.

Comparing the first four lines of tables 2 and 4 shows that for TOH4, compressing a larger PDB into a smaller size by the smallest discs is much better then an original PDB of

3. This does not contradict equation 2 of section 4.3.1. That equation dealt with a configuration that both the compressed and uncompressed PDBs were indexed by the same variables. In our case, however, we compared a compressed PDB that was indexed by discs 2-14 (where disc 1 was compressed) to an uncompressed PDB indexed by discs 1-13. The equation will be valid if we compare an uncompressed PDB of discs 2-14 compared to a our compressed PDB.

PDBs	Discs	Type	Avg h	Nodes	Time	Memory
17-disc problem						
14-3	0	static	90.5	>393,887,912	>421	256M
14-3	0	dynamic	95.7	238,561,590	2501	256M
15-2	1	static	103.7	155,737,832	83	256M
16-1	2	static	123.8	17,293,603	7	256M
18-disc problem						
16-2	2	static	123.8	380,117,836	463	256M

Table 5: Solving the 17 and 18-disc Towers of Hanoi

the same size. For example, comparing the third line of these tables shows that a compressed PDB with 16 megabytes of memory solved the problem more than ten times faster than a simple uncompressed PDB of the same size where the indices are based on the same discs (discs 1-12 in our case). This is empirical evidence for equation 2 of section 4.3.1.

The last row of Table 4 represents lossless compression of the full 14-disc PDB by the smallest of the 14 discs, where we stored one additional bit for each position of that disc. This used $8 + 4 = 12$ bits per four entries instead of $8 \cdot 4 = 32$ bits in the uncompressed PDB (row 1). While the number of generated nodes is identical to row one, the table clearly shows that it is not worthwhile using lossless compression for TOH4 since it requires more time and more memory than lossy compression by the one or two smallest discs (rows 2 and 3).

The *Avg h* column gives the average heuristic over all entries of the PDB. The difference between the average h value of a compressed PDB, and the average h value of the uncompressed PDB (87.04), gives the average loss of information due to compression. The maximum possible loss of information for lossy compression by z discs, the diameter r, is presented in the fourth column of Table 4. This is the length of an optimal solution for a problem with z discs if z is less than 15, since for those problems, two states that are furtherest apart in the problem space are the standard initial and goal states.[4] We observe that the average loss of information is about half the maximum possible information loss d.

To summarize this set of experiments, we conclude that compressing the large discs is not effective because the values of compressed patterns are not highly correlated. By contrast, compressing the small discs is extremely efficient because the values of the compressed entries are highly correlated.

5.4 17 and 18-Disc Problems

We also solved the 17- and 18-disc problems by compressing the smallest discs. The optimal solutions from the standard initial state are 193 and 225 moves respectively. Results are presented in Table 5. Static partitioning of the largest 14 discs and the smallest 3 discs cannot solve the 17-disc problem, since memory is exhausted before reaching the goal after

4. Surprisingly, this is not true with 15 discs, or 20 or more discs (Korf, 2003, 2004).

7 minutes (row 1). With an uncompressed PDB of 14 discs we were only able to solve the 17-disc problem with dynamic partitioning (row 2).

The largest PDB that we could compute entirely in 1 gigabyte of memory was for 16 discs. This database was constructed with a bit-array to detect duplicate nodes, requiring $4^{16} = 4$ gigabits, or half a gigabyte. Given the same amount of memory as the full 14-disc PDB, 256MB, we solved the 17-disc problem in 83 seconds with a 15-disc PDB compressed by the smallest disc, and in 7 seconds with a 16-disc PDB compressed by the smallest two discs. This is an improvement of almost two orders of magnitude compared to row 1. The improvement is 2.5 orders of magnitude compared to the dynamically partitioned heuristic of the 14-disc PDB of row 2. A PDB of 16 discs compressed by 2 discs consumes exactly the same amount of memory as an uncompressed PDB of 14 discs but it is much more informed, as it includes almost all the information about 16 discs. With this PDB we were also able to solve the 18-disc problem in under 8 minutes.

5.5 Using Symmetry and Disk Storage to Solve up to 31 Discs

The algorithms described above are able to find shortest paths between any legal states of TOH4. However, we can do much better if we're only interested in the shortest path from the standard initial state, where all discs are located on one peg, to the standard goal state, where they are all located on another peg (Hinz, 1997). To do this we take advantage of the symmetry between the standard initial and goal states. In particular, we only need to search half way to the goal, to the first *middle* state where all discs but the largest are distributed over the two intermediate pegs. Given such a middle state, we can reach the goal state by moving the largest disc to the goal peg, and then applying the moves made to reach the middle state in reverse order, but interchanging the initial and goal pegs.

The challenge is to take advantage of a heuristic function in this half-depth search to a middle state. The difficulty is that to solve an n disc problem, there are 2^{n-1} middle states, one for each way to distribute the $n-1$ smallest discs over the two intermediate pegs. A PDB heuristic provides a solution to this problem. Rather than building the PDB by a breadth-first search starting with a single goal state, we simply start the search with all 2^{n-1} middle states at depth zero, and search breadth-first until the entire pattern space is generated. The resulting PDB values will be the minimum number of moves required to reach any of the middle states. We refer to this as a *Multiple-Goal Pattern Database*, or MGPDB (Korf & Felner, 2007).

We constructed our heuristic as follows. We first constructed a 22-disc MGPDB compressed to the size of a 15-disc PDB, by compressing the 7 smallest discs. This database occupied exactly a gigabyte of memory, at one byte per entry, and was constructed in memory, using magnetic disk storage for the states of the breadth-first search. We also constructed a separate 8-disc MGPDB, which required very little time or memory. We then used these MGPDBs to compute heuristics for problems with up to 31 discs. For the 31-disc problem, the goal of the search is a middle state where the 30 smallest discs are distributed over the two intermediate pegs. Thus, we don't need to consider the largest disc. For each state of the search, we statically divided the 30 discs into the largest 22 discs, and the smallest 8 discs. We looked up the configuration of the 22 largest discs in the 22-disc MGPDB, we looked up the configuration of the 8 smallest discs in the 8-disc MGPDB, and added the resulting values together. Similarly, we also looked up the configuration of the 22 smallest discs in the same 22-disc MGPDB, and looked up the configuration of the 8 largest discs in the same 8-disc MGPDG, and added theses values together as well. Finally, we took the maximum of these two sums as our overall heuristic.

The search algorithm we used was a frontier version of breadth-first heuristic search (BFHS) (Zhou & Hansen, 2006), with the A* cost function of $f(s) = g(s) + h(s)$, where $g(s)$ is the depth of state s from the initial state, and $h(s)$ is the MGPDB heuristic described above, which estimates the distance to the closest middle state. Using these techniques, and a number of others, we were able to verify the presumed optimal solution for all four-peg Towers of Hanoi problems with up to 31 discs. The 31-disc problem took over 100 CPU-days to run, and used two terabytes of disk storage. Due to an unrecoverable disk error and the resulting loss of a single disk block, there is a one in 200 million probability that there is a shorter solution to the 31-disc problem. We were able to solve all smaller problems with no errors. This represents the current state-of-the-art for the four-peg Towers of Hanoi Problem. The interested reader is referred to (Korf & Felner, 2007) for more details on these experiments.

6. The Sliding-Tile Puzzles

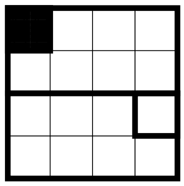

Figure 6: A 7-7-1 partitioning into disjoint sets of the 15-Puzzle

The best method for solving the sliding-tile puzzles optimally uses disjoint additive pattern databases (Korf & Felner, 2002). In this case, variables represent tiles and values represent their locations. The tiles are partitioned into disjoint sets, and a PDB is built for each set. The PDB stores the cost of moving the tiles in the pattern set from any given arrangement to their goal positions. Since for each set of pattern tiles we only count moves of the pattern tiles, and each move only moves one tile, values from different disjoint PDBs can be added together and the results are still admissible. For a deeper analysis of additive PDBs, see (Felner et al., 2004a).

An $x-y-z$ partitioning is a partition of the tiles into disjoint sets with cardinalities x, y and z. Figure 6 shows the 7-7-1 disjoint partitioning of the 15-puzzle used in this paper. The geometric symmetry of this domain can be used to allow another set of PDB lookups (Culberson & Schaeffer, 1998; Korf & Felner, 2002; Felner et al., 2004a). For example, if we reflect the puzzle about the main diagonal, we get another partitioning of this puzzle which is geometrically symmetric to the original partitioning. Therefore, the same PDB can be used to retrieve values for both the regular partitioning and for the reflected partitioning. The maximum of these values can then be taken as an admissible heuristic.

6.1 Combining Functional Heuristics with PDBs

In many domains there exist simple functional heuristics that can be calculated very efficiently. An example is the Manhattan distance (MD) heuristic for the sliding-tile puzzles. In such domains, a PDB can store just the additional increment (Δ) above the functional heuristic, in order to save memory. We denote such a PDB as a DPDB. During the search we add values from the DPDB to the value of the functional heuristic.

For the sliding-tile puzzles we can build a DPDB by storing just the additional increment above MD, which results from conflicts between the tiles in the PDB. These conflicts come in units of two moves, since if a tile moves away from its *Manhattan-distance path*, it must return to that path again for a total of two additional moves. Compressing such a DPDB for the sliding-tile puzzle can be very effective. Consider a pair of adjacent patterns of the sliding-tile puzzle whose values are stored in a DPDB. Since they are adjacent, their Manhattan distances differ by one, but the numbers of additional moves above their MDs are often the same. Thus, much of the information is preserved when compressing the two entries and taking their minimum.

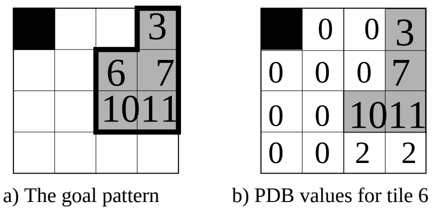

Figure 7: The goal pattern for tiles {3,6,7,10,11} and values for tile 6

For example, consider the subproblem of the 15-puzzle which includes tiles {3,6,7,10,11}. The corresponding goal pattern is shown in Figure 7.a. Assume that all these tiles except tile 6 are in their goal positions, and assume that tile 6 is in location x, which of course cannot be any of {3,7,10,11}. The values in Figure 7.b written in location x correspond to the number of moves above their MDs that the pattern tiles 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 in the bottom row below where tile 10 belongs. In that case tile 6 is in a *linear conflict* (Hansson, Mayer, & Yung, 1992) with tile 10, and one of them must make at least two horizontal moves more than its MD. Thus we write the number 2 in that location. For locations where no moves beyond MD are needed we write 0.

Note that most adjacent positions in Figure 7.b have the same value. Thus, if we build such a DPDB and compress two entries that correspond to such patterns, no information will be lost in most cases. In fact, for the 7-7-1 partition used in the experiments below, when compressing two such patterns into one, we have found that more than 80% of the pairs we compressed stored exactly the same Δ value as before the compression.

6.2 Compressing PDBs for the Sliding-Tile Puzzles

Since the 15 puzzle is a permutation problem, both the *sparse mapping* and the *compact mapping* defined in section 3.1 are applicable. For the 7-tile PDB of our experiments, sparse mapping uses a multi-dimensional array of size 16^7 which has 268×10^6 entries. Alternatively, compact mapping uses an array of size $16 \times 15 \times \ldots \times 10$ which has 57×10^6 entries. Compact mapping is more complex to implement and turned out to be more time consuming in our experiments but needs a smaller amount of memory.

6.2.1 Cliques in the Sliding-Tile Puzzles

In the sliding-tile puzzles a full compression of one variable, or tile, is not equivalent to clique compression. Since every move moves a single tile to an adjacent location, then the largest clique in the pattern space is of size two, or just a single edge. In such edges, the locations of all the tiles of the pattern except one are fixed, and the remaining tile is in one of two adjacent positions. We refer to compressing these two states as *edge compression*.

For the sparse mapping, there are 16 different entries for each variable which correspond to the 16 different possible locations of the tile in the 15-puzzle. In order to take advantage of edge compression, we divided the 16 locations into the following 8 pairs: (0,1), (2,3) ...(14,15). Instead of storing 16 entries for the location of the last tile, we 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 PDB which has 16^7 different entries. Note that compressing these particular edges is logically done by applying $DIV\, 2$ to the index of the last tile.

Compressing edges (cliques) with the compact mapping is more complicated. If the PDB is based on P tiles, there are only $16 - P + 1$ entries for the last tile. For example, if $P = 7$, then out of the 16 different locations, only 10 are legal positions to place the last tile, since 6 are occupied by the other tiles. If we use the same pairing mechanism described above for compressing edges then we can compress the 16 locations to 8 entries. This is only slightly smaller than the 10 entries of the original compact mapping. Edge compressing will only be effective for the compact mapping if the number of pattern tiles is considerably smaller than half the size of the puzzle. For example, in the 24-puzzle, it will be efficient to compress edges for PDBs of 6 tiles even with the compact mapping. Note that an alternative method would be to compress the 10 entries of the compact mapping into 5 entries (by *Div 2*). However, as shown below this will not necessarily compress adjacent locations of the last tile, and therefore will not correspond to edge (clique) compression.

6.3 Results for the 15-Puzzle

Table 6 presents results for different compressing methods on the 15-puzzle for sparse mapping, and Table 7 presents similar results for compact mapping. All the values in the tables are averages over the 1000 random initial states that were used by (Korf & Felner, 2002). The *Heuristic* column defines the partitioning that was used. For example, **1 7-7-1** means that we used one 7-7-1 partitioning. **2 7-7-1** means that we used two 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 next column indicates the compressing method used. When a subscript *ls* is shown it means that lossless compression was used, and otherwise the compression was lossy. The next columns present the number of nodes generated by IDA*, the average running time in seconds, the amount of memory in megabytes at one byte per entry, and the average heuristic of the initial states. The time

No	Heuristic	Compress	Nodes	Time	Mem	Av h
One PDB lookup						
1	1 7-7-1	no	464,977	0.058	524M	43.64
2	1 7-7-1	edge	565,881	0.069	262M	43.02
3	1 7-7-1	edge_{ls}	487,430	0.070	262M	43.59
4	1 7-7-1	row	1,129,659	0.131	131M	42.43
5	1 7-7-1	2 tiles	1,312,647	0.152	131M	42.21
A PDB lookup and its reflection						
6	1+ 7-7-1	no	124,482	0.020	524M	44.53
7	1+ 7-7-1	edge	148,213	0.022	262M	43.98
8	1+ 7-7-1	row	242,289	0.048	131M	43.39
Two different PDBs						
9	2 7-7-1	edge	147,336	0.021	524M	43.98
10	2+ 7-7-1	edge	66,692	0.016	524M	44.92

Table 6: 15-puzzle results. Different compressing methods for one or more lookups with sparse mapping

needed to precompute the PDB is traditionally omitted, since one only needs to precompute it once, and the same PDB can be used to solve as many problem instances as needed.

6.3.1 Sparse mapping

Row 1 of Table 6 presents the benchmark results for the single 7-7-1 partitioning with no compression.[5]. The next four rows (2-5) present results for different compression methods for this 7-7-1 PDB. Row 2 gives the results of the 7-7-1 PDB where the two 7-tile PDBs were compressed by edges. While the size of the PDB was cut in half, the overall effort was increased by no more than 20% in both the number of generated nodes and in overall running time. Row 3 presents results of the same 7-7-1 partitioning when we used lossless compression of edges. While the number of generated nodes decreased by 15% from the lossy compression, the overall time increased a little. This is due to the additional constant time for handling the lossless compression.[6] Row 4 provides results for the case where the 16 different locations of one tile were compressed into four different entries by saving only the row of the tile's position, and not its column. This can be logically done by *DIV 4*. In both cases of moving from row 1 to row 2 (in Table 6) and from row 2 to 4, the amount of memory

5. The best results for the 15 puzzle are achieved by using a 7-8 partitioning (Korf & Felner, 2002). This partitioning cannot be implemented with sparse mapping as it would need 4 Giga bytes of memory. Thus, we used a 7-7-1 partitioning for this set of experiments.
6. The reason that the number of generated nodes was not identical to the 7-7-1 partitioning without compression is because in rare cases in our PDBs, the two entries we compressed were not edges and differed by more than one move. Thus, information was lost even with the lossless compression that we used. These rare cases are caused by the special way that we treated the location of the blank. A full technical treatment how the blank is handled is provided by (Felner et al., 2004a).

was reduced by a factor of two. However, while the number of generated nodes increased by 20% by edge compression, the number of generated nodes increased by a factor of 2.5 when going from edge compression to row compression. The reason is again the correlation among the values of the compressed entries. With edge compression the loss of information is at most one move, and this does not significantly effect the overall performance. With row compression, the loss of information is much greater because the four values compressed can be significantly different from each other. Thus, the effect on overall performance is much greater. Row 5 represents an alternative way to compress the PDB by a factor of 4. In this case, the locations of two tiles were compressed into eight locations each. Results were a little worse than for row compression, because the correlation among the compressed values was even less in this case.

The next three rows (6-8) show the same tendency when two sets of PDB lookups were performed on the 7-7-1 PDB, the original set and the same set reflected about the main diagonal. Row 6 presents the results for the uncompressed versions of these PDBs. Note that an additional PDB lookup from the same PDB of row 1 improved the results by a factor of almost four. Again when compressing edges in row 7, the correlation among the compressed values is high, the loss of information is at most one move, and the running time is roughly the same compared to no compression. With row compression in row 8, the correlation of values is worse, and the search effort increases.

Edge compression causes a small loss of information but reduces the memory by half. Thus, we can use the same amount of memory as the original uncompressed PDB but for two compressed PDBs. Row 9 presents results when we took two different 7-7-1 PDBs, compressed them by a factor of two using edges, and took their maximum. This configuration uses the same amount of memory as the benchmark uncompressed single 7-7-1 partitioning of row 1, but solves the problem almost three times faster. Row 10 also computes the reflection about the main diagonal of these two different compressed PDBs and takes the maximum of the 4 different partitionings. This reduced the number of generated nodes by a factor of 7 and a factor of 2, compared to the uncompressed versions with the same amount of memory of row 1 and 6, respectively. In terms of running time the speedup was more modest and was further decreased to 16 milliseconds. This is because we now have 4 PDB lookups and there is a diminishing return in adding more lookups.[7]

6.3.2 Compact mapping

Table 7 presents results where the PDB was built by compact mapping. The first line provides results for the same 7-7-1 partitioning used for the sparse mapping of table 6 but with compact mapping. The indexing algorithm used for this line is a simple algorithm where calculating the exact index takes time that is quadratic in the number of objects in the pattern. Note that the number of generated nodes here is identical to line 1 of table 6 but the amount of memory needed is much smaller. On the other hand, our results show that the actual CPU time for the compact mapping is worse than the corresponding sparse mapping. In the rest of the table we used the advanced algorithm for index calculation of

7. See (Holte, Felner, Newton, Meshulam, & Furcy, 2006) for a deeper discussion of this and for advanced methods to further reduce the constant time per node when a number of PDB lookups are performed. Furthermore, the exact CPU time measured in all our experiments should be taken with care since it is greatly influenced by the implementation, compiler and the hardware of the machine used.

No	Heuristic	Compress	Nodes	Time	Mem	Av h
Simple compact mapping						
1	1 7-7-1	none	464,977	0.232	55M	43.64
Advanced compact mapping						
2	1 7-7-1	none	464,977	0.121	55M	43.64
3	1 7-7-1	edge	565,881	0.142	44M	43.02
4	1 7-7-1	edge $_{ls}$	487,430	0.130	44M	43.59
5	1 7-7-1	last tile	996,773	0.240	27.5M	42.87
6	1 7-7-1	first tile	1,024,972	0.261	27.5M	42.94

Table 7: 15-puzzle results. Compressing the compact mapping

compact mapping that was presented by (Korf & Shultze, 2005). In this advanced algorithm the time to calculate the exact index is reduced to linear in the number of objects. This is done with the help of another lookup table which stores values of shift operations needed by the algorithm. See (Korf & Shultze, 2005) for a deeper discussion and treatment of this method. Using the advanced algorithm further reduced the running time by a about a factor of two over the simple quadratic algorithm but this is still slower than the corresponding sparse mapping reported in table 6.

Lines 3 and 4 provide results for lossy and lossless edge compressing with the advanced compact mapping. The results report the same number of generated nodes as the corresponding sparse mapping from table 6. Compact mapping only has 10 entries for the last tile (since 6 locations were occupied by the other tiles). Edge compression (lines 3 and 4) slightly reduced this to 8 entries offering a small memory reduction of only 20% over the uncompressed 7-7-1 compact mapping PDB. This is probably not cost effective and it will not allow another compressed PDB to be stored in the same amount of memory as the uncompressed PDB. Line 5 presents results where we compressed the index of the last tile from 10 entries into 5 by taking the maximum of two adjacent entries. This is equivalent to *Div 2*. The memory reduction here is a factor of 2. However, since neighboring entries in the compact mapping do not necessarily correspond to cliques, the number of generated nodes significantly increased as the loss of information is not guaranteed to be at most 1 as in edge (clique) compressing. The last line provides similar results where the compressing was performed on the index of the first tile (from 16 locations to 8). This is equivalent to *MOD 2*. Again, the entries that were compressed are not cliques and the loss of data was rather large.

6.3.3 Summary of the 15 puzzle results

Our results on the 15 puzzle show that only edge (clique) compressing is effective on this puzzle and most of the information is preserved by this type of compressing. Other compressing techniques cause significant loss of information and are not efficient on this domain. Edge compressing provides a significant memory reduction of a factor of 2 if sparse mapping is used. This is encouraging since sparse mapping will probably be the best choice for larger versions of the puzzle (e.g., 24, 35, etc.). If one chooses to use the compact mapping then memory saved by edge compressing is rather modest and probably not effective. Unlike the 4-peg Towers of Hanoi problem, we could not find a compressing method with a memory saving factor larger than 2 that proved efficient in the tile puzzle.

The best existing heuristic for the 15 puzzle is a PDB based on a 7-8 partitioning of the tiles, and its reflection about the main diagonal (Korf & Felner, 2002). We solved the problem by implementing this partitioning with compact mapping which used 562 megabytes. The average number of generated nodes was 36,710, and the average running time was .018 seconds for fast compact mapping. This is a reduction of almost a factor of two over the simple quadratic compact mapping. Since the 7-8 PDBs are implemented by compact mapping we could not improve these results with compression. It is worth noting that our best version of the 7-7-1 partitioning (line 10 of Table 6) uses slightly less memory (524 megabytes) and runs slightly faster(.016 seconds), but generates twice as many nodes. (66,692).

6.4 Results for the 24-Puzzle

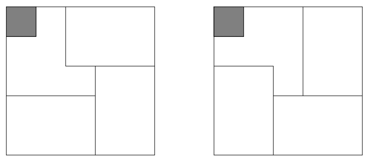

Figure 8: The 6-6-6-6 PDB for 24-Puzzle and its reflection

The best existing heuristic for the 24-puzzle when only one gigabyte of memory is available is the 6-6-6-6 partitioning and its reflection about the main diagonal (Korf & Felner, 2002) shown in Figure 8. We compressed the same 6-6-6-6 partitioning and found that similar to the 15 puzzle lossy compression of edges was effective and only generated about 20% more nodes. However, by 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 set of 6-6-6-6 partitioning, even without any compression, only achieves a small reduction in node generations.

We also tried a 7-7-5-5 partitioning and its refection, which were stored in one gigabyte of memory by compressing the 7-tile PDBs. Even without compression, the number of generated nodes was not much better than the 6-6-6-6 partitioning which is probably the best 4-way partitioning of this puzzle.

One way to obtain a speedup in this domain might be to compress larger PDBs such as an 8-8-8 partitioning, but that is beyond the scope of this work. An 8-8-8 PDB was implemented by (Felner & Adler, 2005) using a sophisticated method of instance-dependent PDBs. This method is based on an idea first presented by (Zhou & Hansen, 2004), in which only the relevant parts of the PDB are stored in memory given particular initial and goal states.

7. Top-Spin

We have tried different compression methods on the Top-Spin domain as well. In Top-Spin more than one object is moved in each move. Therefore, simple disjoint additive PDBs are not applicable here. The simple way to build a PDB for this domain is to specify a number of pattern tokens, and for the remaining tokens to be indistinguishable. Here, we only used the compact mapping.

7.1 Cliques in Top-Spin

Due to the nature of Top-Spin, the largest cliques are of size two, or simply edges, but unlike the other domains they do not correspond to compression of single variables. For example, assume a 3-token PDB of tokens $(2, 3, 4)$ for the (9,4) Top-Spin problem. Note that patterns $(*, 2, 3, 4, *, *, *, *, *)$ and $(*, *, 4, 3, 2, *, *, *, *)$ are adjacent, and therefore belong to an edge clique of size two. However, all three tokens move here, and hence all three pattern variables change their values. Thus, compressing edges (cliques) is not as simple as compressing the value of a single variable in this problem.

7.2 Compression Based on Individual Variables

In fact, in Top-Spin, compressing by individual variables is the same as ignoring those variables. In other words, a PDB based on P variables, compressed by C variables, is actually identical in both memory and values to a PDB based on $P - C$ variables.

To explain this, we first examine compression by a single variable. Consider a three-variable PDB_3 based on tokens $(1, 2, 3)$, and a four-variable PDB_4 based on tokens $(1, 2, 3, 4)$ for the (9,4) Top-Spin problem. The non-pattern tokens are represented by $*$. The goal pattern for PDB_3 is $(1, 2, 3, *, *, *, *, *, *)$, and the goal pattern for PDB_4 is $(1, 2, 3, 4, *, *, *, *, *)$. Let $p_1 = (1, 2, *, *, *, *, *, *, 3)$ be an example pattern in PDB_3. We can reach p_1 from the goal pattern in two moves, and since the operators in this space are their own inverses, $PDB_3(p_1) = 2$. If we apply these same two moves to the pattern $(1, 2, 3, 4, *, *, *, *, *)$, we reach the pattern $(1, 2, *, *, 4, *, *, *, 3)$, which we'll call p_2. Thus, $PDB_4(p_2) = 2$. Now consider PDB_{4-1} which is PDB_4 compressed by token or variable 4. By definition, $PDB_{4-1}(p_1)$ is the minimum value over all locations for token 4, of $PDB_4(p_i)$. In fact, we constructed p_2 such that $PDB_4(p_2) = PDB_3(p_1)$. Since there can be no smaller value, $PDB_{4-1}(p_1) = PDB_4(p_2) = PDB_3(p_1)$. The same argument applies to any other example pattern. Furthermore, we can apply the same argument to a PDB built by compressing over any number of individual variables. Thus in the special case of Top-Spin, equation 2 from section 4.3.1 becomes $CPDB_{P/C}(s) = PDB_{P-C}(s)$. This means that for Top-Spin, compression based on individual variables offers no benefits at all compared to simply ignoring those variables in a PDB of the same size.

7.3 Experimental Results for Top-Spin

Since compression based on individual variables is of no benefit here, we tried general compression by applying the DIV and MOD operators to the PDB indices. We experimented with the (17,4)-Top-Spin problem and started with a compact mapping PDB of 9 consecutive tokens, including the 1 token. Note that with a few exceptions noted below, these

Comp. Factor	Avg. Value	Avg Nodes	Avg Time	Size of PDB
9-token PDB				
1	10.52	40,810,940	87.36	495M
2	10.20	59,827,209	127.54	247M
3	10.03	87,517,365	183.70	164M
4	9.88	86,424,249	184.19	123M
5	9.76	127,276,981	264.33	99M
6	9.69	147,626,798	307.42	82M
7	9.61	128,757,535	267.73	71M
8	9.54	159,711,937	331.97	62M
9	9.53	313,375,790	650.83	55M
8-token PDB				
1	9.53	313,375,790	650.83	55M

Table 8: Results for (17,4) Top-Spin with PDBs of 9 and 8 tokens where the 9-token PDB is compressed by *DIV* operator

Comp. Factor	Avg. Value	Avg Nodes	Avg Time	Size of PDB
1	10.52	40,810,941	87.37	495M
2	10.13	50,363,034	109.88	247M
3	9.89	57,576,194	119.86	164M
4	9.82	76,123,453	158.47	123M
5	9.65	87,573,074	183.05	99M
6	9.53	87,356,615	186.15	82M
7	9.38	85,280,514	205.25	71M
8	9.49	152,480,885	321.23	62M
9	9.17	89,543,322	197.18	55M

Table 9: Results for(17,4) Top-Spin with a 9-token PDB compressed with *MOD*

compressions do not preserve the structure of the state variables and there is no correlation of values between the compressed entries.

Table 8 presents the results of *DIV* compression performed on our PDB. Each line represents a different compression factor, which is the second argument to the *DIV* operator. As expected, a larger compression factor increased the search effort, but reduced the amount of memory. Note that the *DIV* 9 corresponds to compression based on a single variable, since once 8 variables have been set in a 17-variable permutation problem, there are exactly 9 possible locations remaining for the next variable. Thus, as explained in the previous section this line is identical to an uncompressed PDB of size 8 (the last line of the table).

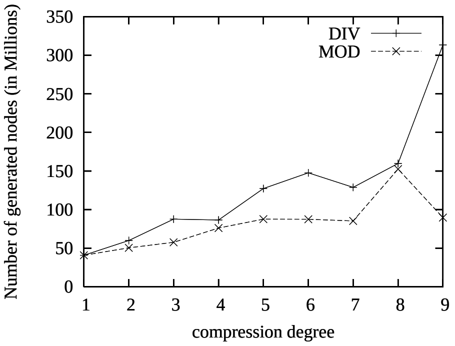

Figure 9: Nodes generated by *DIV* and *MOD* for the 9-token PDB of the (17,4)- Top-Spin problem

Table 9 shows the results of compressing the same PDB with the *MOD* operator, in the same form as Table 8. Figure 9 compares the two methods. Surprisingly, *MOD* outperformed *DIV*. This is counterintuitive because *DIV* compresses entries that tend to have many state variables in common. Our best explanation for this phenomenon is as follows. As described above, in Top-Spin the location of several tokens is changed in a single move. We believe that the distance between two states that are similar, such as differing by a swap of two tokens for example, is greater than the distance between a randomly-chosen pair of states.[8] Thus, states that are grouped together by the *DIV* operator will tend to be less highly correlated than random groupings, or groupings induced by the *MOD* operator.

Note that the last line of Table 9 (compression by *MOD* 9) used the same amount of memory as an uncompressed 8-token PDB (last line of Table 8) but takes only 30% as long to solve the problems. This shows the benefit of compression in this problem as well. Compression by the *MOD* operator is effective and for the given size of memory of the 8-token PDB, it is better to build a larger 9-token PDB and compress it to that size. This reduces the search effort by a factor of 3.

8. Conclusions

We introduced a new method for improving the performance of PDBs, by compressing larger PDBs to fit into smaller amounts of memory. We applied the technique to the four-peg Towers of Hanoi problem, the sliding-tile puzzles, and the Top-Spin puzzle. Our experiments confirm that given a specific amount of memory M, it is usually better to use this memory with compressed PDBs than with uncompressed PDBs. This can be practically achieved either by a larger PDB compressed to size M, or by maximizing over k compressed PDBs each of size M/k.

8. We performed several experiments to confirm this. Indeed, for the (12,4) - Top-Spin version, a pair of random states are distanced 9.28 moves away from each other on average while two states that only differ in two adjacent tokens are 12 moves away. Similarly, for the (16,4) - TopSpin the average distances were 14.04 and 16 moves, respectively.

We introduced a number of different methods for compressing a PDB, ranging from very general mapping functions of the indices such as *DIV* and *MOD*, to methods which take into account the structure of the problem space. In general, we want to group together PDB entries that have similar values. This can be achieved by grouping patterns that are close to each other in the pattern space. In particular, states that form a clique in the problem space will have PDB values that differ by at most one, and hence are natural candidates for compression. The exact method for finding these cliques (or nearby patterns) in the PDB depends on the domain and on the exact way that the PDB is implemented.

For TOH4 and for the sparse mapping of the tile puzzle, a PDB can be constructed very easily in such a way that cliques reside in nearby entries of the PDB and thus, the PDB values are locally correlated. Therefore, compressing nearby PDB entries proved useful for these settings. For Top-Spin, however, values of a PDB stored in the standard way are not locally correlated and thus compressing nearby PDB entries was not effective. In this domain it is effective to compress patterns that are far apart in the PDB. Similarly, neighboring entries for the compact mapping of the tile puzzle do not correspond to cliques. Thus, compression did not prove useful in this setting and we could not improve the 7-8 partitioning of the 15 puzzle.

In the Towers of Hanoi problem, we achieved dramatic improvements of several orders of magnitude in running time, compared to uncompressed PDBs of the same size, and used this technique in verifying the optimal solutions for problems with up to 31 discs, which is the current state of the art. For the sliding-tile puzzles we showed that compression can preserve most of the information and our techniques offer some practical improvements for the sparse mapping but not for the compact mapping. For the Top-Spin, we achieved improvements with a very naive compression method (the *MOD* method).

We also described several methods for generating PDBs that are too large to fit into memory prior to compression, by using auxiliary disk storage.

9. Acknowledgements

This research was supported by the Israel Science Foundation (ISF) grant No. 728/06 to Ariel Felner. It was also supported by NSF grant No. EIA-0113313 to Richard Korf.

References

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13(3)*, 422–426.

Chen, T., & Skiena, S. (1996). Sorting with fixed-length reversals. *Discrete Applied Mathematics*, *71*(1-3), 269–295.

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

Dunkel, O. (1941). Editorial note concerning advanced problem 3918. *American Mathematical Monthly*, *48*, 219.

Dunkel, O. (1992). Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, *24(3)*, 142–170.

Edelkamp, S. (2001). Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning (ECP-01)*, pp. 13–34.

Edelkamp, S. (2002). Symbolic pattern databases in heuristic search planning. In *Proc. International Conference on AI Planning and Scheduling (AIPS)*, pp. 274–293.

Edelkamp, S., & Kissmann, P. (2007). Externalizing the multiple sequence alignment problem with affine gap costs. In *German Conference on Artificial Intelligence (KI), LNCS 4467*, pp. 444–447.

Felner, A., & Adler, A. (2005). Solving the 24-puzzle with instance dependent pattern databases. In *Proceedings of SARA-05*, pp. 248–260, Edinburgh, Scotland.

Felner, A., Korf, R. E., & Hanan, S. (2004a). Additive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, *22*, 279–318.

Felner, A., Meshulam, R., Holte, R., & Korf, R. (2004b). Compressing pattern databases. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pp. 638–643.

Felner, A., Zahavi, U., Holte, R., & Schaeffer, J. (2005). Dual lookups in pattern databases. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pp. 103–108.

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

Hansson, O., Mayer, A., & Yung, M. (1992). Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, *63(3)*, 207–227.

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

Hinz, A. M. (1997). The tower of Hanoi. In *Algebras and Combinatorics: Proceedings of ICAC'97*, pp. 277–289, Hong Kong. Springer-Verlag.

Holte, R. C., Felner, A., Newton, J., Meshulam, R., & Furcy, D. (2006). Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, *170*, 1123–1136.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27(1)*, 97–109.

Korf, R. E. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pp. 700–705.

Korf, R. E. (2003). Delayed duplicate detection: Extended abstract. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pp. 1539–1541, Acapulco, Mexico.

Korf, R. E. (2004). Best-first frontier search with delayed duplicate detection. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004)*, pp. 650–657, San Jose, CA.

Korf, R. E., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, *134*, 9–22.

Korf, R. E., & Felner, A. (2007). Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 2324–2329.

Korf, R. E., & Shultze, P. (2005). Large-scale, parallel breadth-first search. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*, pp. 1380–1385, Pittsburgh, PA.

Korf, R. E., Zhang, W., Thayer, I., & Hohwald, H. (2005). Frontier search. *Journal of the Association for Computing Machinery (JACM)*, *52*(5), 715–748.

McNoughtton, M., Lu, P., Schaeffer, J., & Szafron, D. (2002). Memory efficient A* heuristics for multiple sequence alignment. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pp. 737–743.

Mero, L. (1984). A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, *23*, 13–27.

Munagala, K., & Ranade, A. (1999). I/o complexity of graph algorithms. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, pp. 687–694. ACM-SIAM.

Myrvold, W., & Ruskey, F. (2001). Ranking and unranking permutations in linear time. *Information Processing Letters*, *79*, 281–284.

Schroedl, S. (2005). An improved search algorithm for optimal multiple-sequence alignment. *Journal of Artificial Intelligence Research (JAIR)*, *23*, 587–623.

Stern, U., & Dill., D. L. (1995). Improved probabilistic verification by hash compaction. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp. 206–240.

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

Zahavi, U., Felner, A., Holte, R., & Schaeffer, J. (2006). Dual search in permutation state spaces. In *Proceedings of the Twenty First National Conference on Artificial Intelligence (AAAI-06)*, pp. 1076–1081.

Zahavi, U., Felner, A., Schaeffer, J., & Sturtevant, N. (2007). Inconsistent heurstics. In *Proceedings of the Twenty Second National Conference on Artificial Intelligence (AAAI-07). To appear.*

Zhou, R., & Hansen, E. (2004). Space-efficient memory-based heuristics. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pp. 677–682.

Zhou, R., & Hansen, E. (2006). Breadth-first heuristic search. *Artificial Intelligence*, *170*(4-5), 385–408.