Compressing Pattern Databases with Learning

Mehdi Samadi[1] and **Maryam Siabani**[2] and **Ariel Felner** [3] and **Robert Holte**[1]

Abstract. A pattern database (PDB) is a heuristic function implemented as a lookup table. It stores the lengths of optimal solutions for instances of subproblems. Most previous PDBs had a distinct entry in the table for each subproblem instance. In this paper we apply learning techniques to compress PDBs by using neural networks and decision trees thereby reducing the amount of memory needed. Experiments on the sliding tile puzzles and the TopSpin puzzle show that our compressed PDBs significantly outperforms both uncompressed PDBs as well as previous compressing methods. Our full compressing system reduced the size of memory needed by a factor of up to 63 at a cost of no more than a factor of 2 in the search effort.

1 Introduction and Overview

States in a search space are often represented using a set of state variables. An abstraction of the search space, called the *pattern space*, can be defined by only considering a subset of the state variables (called the *pattern variables*). A *pattern* is a state of the pattern space which has an assignment of values to the pattern variables. A state s in the original space is mapped to a pattern s' by ignoring the state variables in s that are not pattern variables. A *pattern database* (PDB) stores the distance of each pattern to the goal pattern. The value stored in the PDB for s' is a lower bound on the distance from s to the goal state, and thus serves as an admissible heuristic for searching in the original search space.

A PDB contains one entry for each pattern in pattern space. In general, the more entries a PDB contains, the more accurate it is as a heuristic, and the more efficient is the search that uses the PDB as a heuristic. The drawback of large PDBs is the amount of memory they consume. One approach to mitigating this problem is to *compress* the PDB. For example, Felner et al. [3] compress a PDB by simply merging several highly correlated (usually adjacent) entries into one. They achieved a significant improvement on the 4-peg Towers of Hanoi and the TopSpin problems but only limited success for the sliding tile puzzles. The main drawback of that work is that the rule for deciding which PDB entries to merge was fixed throughout the entire compressing process. Higher degree of compression significantly degrades the performance.

We introduce a new, general and flexible compression method for PDBs that is experimentally shown to improve uncompressed PDBs as well as the compression methods reported in [3]. Improvement takes the form of either reducing the amount of memory required for the PDB without substantially increasing the number of generated nodes, or reducing both the memory required and the number of generated nodes. The main idea underlying our work is to use techniques from the machine learning literature to compress PDBs. In particular, we train an artificial neural network (ANN) so that it can be used instead of the PDB. The neural network requires almost no memory. However since the ANN's output is not guaranteed to be less than or equal to the PDB value (admissible), we use additional storage (in the form of a hash table) for all the patterns whose value is overestimated by the ANN. This basic idea is then improved with two steps. Decision trees and a PDB-partitioning method are used to separate the PDB entries into smaller subgroups with similar characteristics and then training separate ANNs for each subgroup.

We tested our compression system on three search spaces: the 15-puzzle, the 24-puzzle and TopSpin. Our results show that our full compression system requires up to 63 times less memory than the original PDB while increasing the number of nodes generated by no more than a factor of two. The modest increase in search effort is not a concern because the freed-up memory can be used in ways that are known to substantially speed up search, e.g., for additional PDBs [6], and/or for memory-based search algorithms such as A*, perimeter search or memory-enhanced IDA*. We do not actually implement any of these techniques in this paper, but are confident that they would more than compensate for the small increase in search effort caused by our compression technique.

2 Related Work

Symbolic PDBs [1] use binary decision diagrams (BDDs) to store a PDB, and have been shown, for some search spaces, to significantly reduce the memory needed to store the PDB entries compared to traditional PDB tables. However, a recent unpublished study of symbolic PDBs on a wide range of search spaces has shown that symbolic PDBs do not always result in compression, they sometimes require more memory than a table. In particular, symbolic PDBs for the 15-puzzle require more memory than the traditional PDB representation, whereas the experiments below show that our method greatly reduces the memory required.

The idea of using learning and classification techniques in heuristic search was suggested before. In [9], a feature vector for partitioning the state space to a number of classes is used. Learning techniques were used for each class. In [10] the state space was partitioned based on feature vector and then "generalized heuristic information" is learned for each class. These ideas were only applied to small domains and in contrast to our approach did not find the optimal solution.

Recently, multi-layered ANN was used to represent heuristics for the 15-puzzle [2]. Given a training set of states descriptions together with their optimal solution, a learning system that predicts the length of the optimal solution for an arbitrary state was built. They biased their predicted values towards admissibility but unlike our approach their system returned suboptimal solutions in about 50% of the cases.

[1] Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8, {msamadi,holte}@cs.ualberta.ca

[2] Electrical and Computer Engineering Department, Isfahan University of Technology, Isfahan, Iran, siabani@ec.iut.ac.ir

[3] Information Systems Engineering Dept.,Deutsche Telekom Labs,Ben Gurion University,Beer-Sheva, Israel, felner@bgu.ac.il

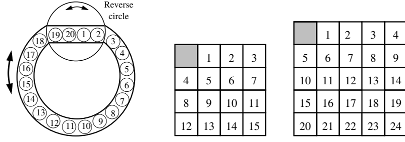

Figure 1. The Top-spin and Sliding tile puzzles.

3 Search domains

The sliding tile puzzles such as the 15- and 24- puzzles (shown in Figure 1) have been used as benchmark problems in many previous papers. For clarity, we describe all our methods in the context of the sliding tile puzzle, but our ideas are general and can be applied to other problems as well. In our representation of the sliding tile puzzle, the variables are the tiles and the values are their locations. The best existing method for solving the sliding tile puzzles optimally uses additive PDBs [4]. 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, counting the moves of the pattern tiles. Under such circumstances the sum of the values from different disjoint PDBs is an admissible heuristic [4]. We use the notation $x - y - z$ to denote a partitioning of the tiles into three disjoint groups with x, y and z tiles in each group, respectively.

The N-TopSpin puzzle has N tokens arranged in a ring. Any set of 4 consecutive tokens can be reversed (rotated 180 degrees in the physical puzzle). Our encoding of this puzzle has N operators, one for each possible reversal. In TopSpin more than one object is moved in each move and simple additive PDBs are not applicable here. The standard way to build a PDB for this domain is to specify a set of pattern tokens, and to treat the remaining tokens as if they were indistinguishable from one another[4].

4 Augmented compression

We introduce a method that compresses PDBs using learning techniques and preserves the admissibility property. Our system includes three independent steps, ANN learning , Decision tree classification and pattern partitioning and we describe each of them in turn.

4.1 Compression with ANN

Our first idea is to build an ANN that learns the PDB. Assume a PDB for the tile puzzle which consists of the set of tiles, S. The different patterns are the different ways to permute all the tiles in S into the state space. Each pattern t has an entry $PDB(t)$ which stores its heuristic value. We want to build a learning system that will be able to predict $PDB(t)$ for each given pattern t. For this we use Artificial Neural Networks (ANN) [8], a well-known learning technique. Multi Layer Perceptron (MLP) neural network (ANN) with standard modified back-propagation [8] algorithm is used for the prediction. This system is called the *basic ANN* compressing in this paper.

4.1.1 Feature selection

We use two types of features for the ANN:

1) Description of the pattern: Each tile in S is a feature and its position in pattern t is the value for that feature.

[4] Since this puzzle is cyclic, we can assume that token number 1 is always in fixed position. Thus, for implementation, the total number of states can be reduced by a factor of N.

2) Heuristic vector: We also construct K smaller PDBs each for a subset of tiles $S_i \subset S$. We denote the corresponding PDB heuristic for a given pattern t as $h_i(t)$. Note that each $h_i(t)$ is admissible for t. We define a *heuristic vector* for pattern t as $H(t) = \langle h_1(t), h_2(t), \ldots, h_K(t)\rangle$. Each member of the heuristic vector h_i is also used as a feature and $h_i(t)$ is the value of that feature. For example, for a 6-tile PDB we used two different 2-4 partitionings to a total of 4 smaller PDBs that are used in the heuristic vector.

The heuristic value of t in the original PDB is the target function.

4.1.2 Training and using the ANN

The ANN is trained by iterating over all the entries of the original PDB. For each pattern t we construct its different features and feed them to the ANN coupled with $PDB(t)$ (the PDB heuristic for t). Once the training process ends, we can delete the original PDB from memory. Only the smaller PDBs which make up the heuristic vector are left in memory. Similarly, the ANN itself is also kept in memory. Then, during the search, given a pattern t, we calculate its features (e.g., by looking up the smaller PDBs). The values of these features are given an input to the ANN and the output ,denoted as $ANN(t)$, is used as the heuristic value for pattern t.

4.1.3 Correcting overestimations

Training an ANN to predict the exact desired value is NP-complete [7]. Thus, learning systems in general and ANNs in particular are not completely accurate by nature as they can deviate from the real value for many of the instances. With heuristics, lower deviations are not a problem as an admissible heuristic should be a lower bound. However, if the ANN is overestimating, the heuristic will no longer be admissible and non-optimal solutions might be returned.

We solved this problem as follows. After the ANN was built, we iterated again on the entire set of patterns (as a test set). Each pattern t whose $ANN(t) > PDB(t)$ is inserted into a hash table H together with its correct heuristic value $PDB(t)$. During the search, we first check to see whether $t \in H$. If indeed $t \in H$ we use its heuristic value stored in H and do not even consult the ANN. As shown below, for well trained ANNs the set of overestimating patterns is small and so is the memory requirements of H.

Traditionally, the training phase is stopped when the *mean square error* (MSE) of the training data is below a predefined small threshold. For our case it is defined as: $MSE = \Sigma_{t \in T}(E(t)^2)/|T|$, where T is the training set, $E(t) = \phi(t) - PDB(t)$, PDB is the original PDB value, and ϕ is the learned function.

$E(t)^2$ is symmetric, so overestimation and underestimation have the same cost. Using this function with an ANN results in a heuristic that tries to be close to the optimal value without discriminating between being under (acceptable) or over (undesirable).

We modified the error function to penalize positive values of $E(t)$ (overestimation), biasing the ANN towards producing admissible values. We used $E'(t) = (a + \frac{1}{1+\exp(-bE(t))})E(t)$ instead of $E(t)$ in the MSE calculations. The constants a and b were determined experimentally. $E'(t)$ reduces the number of overestimating instances by a factor of 4 (over $E(t)$) and was used in our experiments.

4.1.4 Experimental results

In this section, we evaluate our compression system on the 6-6-3 (of tiles (4-9), (10-15) and (1-3)) additive PDB for the 15-puzzle; additional evaluation of the final system is given in Section 5. The compression technique is applied to the two 6-tile PDBs individually;

Heuristic	AvH	Nodes	Time	Mem	Hash
6-6-3	40.06	6,323,187	2.39	11.00	-
$(4\text{-}2)^2$-$(4\text{-}2)^2$-3	37.84	50,818,284	19.71	0.18	-
DIV 2	38.88	19,204,184	7.92	5.50	-
basic ANN	39.20	11,676,726	10.44	1.30	8%
ANN+DT	39.75	9,550,754	5.42	0.84	4%
ADP	39.90	7,285,207	4.62	0.50	2%

Table 1. Results for the 6-6-3 PDBs of the 15-puzzle.

the 3-tile PDB is very small and is left uncompressed. The heuristic vector for each 6-tile PDB contains four values, which are created by using two sets of additive 4- and 2-tile PDBs. Table 1 shows the results. All the values shown are averages over the first 100 random initial states used in [4]. The first column is the heuristic used. The next four columns present the average initial heuristic value, the number of nodes generated by IDA*, the average time (in seconds), and the amount of memory used (in Megabytes). The time needed to precompute the PDB and train the ANN is not included in the times reported. This is standard, since these operations are done just once, no matter how many problems are solved. The final column shows the percentage of entries in the original PDB that were stored in the hash tables because the ANNs overestimated their value.

The first row presents the results of using the normal 6-6-3 PDB. The second row shows the results of directly using the PDBs that make up the heuristic vector inside our ANN system. The superscript 2 in the heuristic description indicates the use of two sets of 4-2 additive PDBs for each 6-tile PDB in the 6-6-3. The maximum value of the two sets is used instead of the 6-tile PDB value. The third row shows the results of using the $DIV2$ method for compressing the 6-tile PDBs used in [3]. In this method, adjacent PDB entries are replaced by a single entry. The fourth row (basic ANN) is for the ANN system just described. The total memory for this system is dominated by the memory needed for the hash table; the memory needed for the small PDBs used in the heuristic vector is small (reported in row 2), and the memory needed for the ANN itself is negligible. The last two rows are for the enhanced ANN systems described below. Again, the total memory needed for them is mostly needed by the hash tables.

The direct use of the smaller PDBs that make up the heuristic vector of our ANN (row 2) dramatically reduces the memory but increases the number of generated nodes by an order of magnitude. By contrast, our basic ANN technique reduces the amount of memory by a factor of 9 while increasing the number of generated nodes by a factor of only 1.84. This is a significant improvement over the 2-fold memory reduction of $DIV2$ compressing technique [3] which was achieved at the cost of 3 times more generated nodes[5].

In all the results of this paper the constant CPU time per node favors simple PDB construction. While we efficiently implemented all our learning techniques, they could probably be made more efficient and better optimized. We decided to also report the CPU time but it should be taken with care.

4.2 Using decision tree to classify data

A major problem of using ANN for predicting the value of PDBs is the size of the hash table used to store the patterns with overestimating ANN values. To address this we first construct a decision tree (DT) which classifies the patterns into two types. The ANN is only used for one type while the other type will consult smaller PDBs.

As described above, each PDB is partitioned into smaller disjoint PDBs. For example a 6-tile PDB, PDB_6 is partitioned into two disjoint PDBs - PDB_2 and PDB_4. We want to classify the 6-tile patterns into two classes: *equal* and *larger*. A pattern t is classified as *equal* if $PDB_6(t) = PDB_2(t)+PDB_4(t)$. It is classified as *larger* if $PDB_6(t) > PDB_2(t) + PDB_4(t)$. Patterns in the *equal* class need only to consult the smaller 2- and 4-tile PDBs and add their values. For patterns in the *larger* class PDB_6 has knowledge about additional moves (over the sum of the two smaller PDBs) that are needed. Thus, the ANN is built to learn these additional moves.

The benefit of using the DT before the ANN is twofold. First, it is sufficient to train and use the ANN for patterns in the *larger* group only. Thus, the ANN can be made more accurate as it needs to learn the behavior of a special class of patterns only - the ones whom their PDB values were larger than the sum of the smaller PDBs. Second, for the *equal* group, there is no need to pass through the complex network of the ANN and consulting the smaller PDBs is enough. Note that deepening down a decision tree is rather cheap as it is usually implemented as a series of nested $if - then - else$ statements.

Adding the DT proved useful. For example, for the 6-6-3 PDBs of the 15 puzzle, nearly 58% of the patterns were classified as *equal*[6] and only 42% are *larger* patterns that trained the ANN.

4.2.1 Building the Decision Tree

A decision tree is built by examining various attributes of the training data. The entire set of features used by the ANN (described above) were used as attributes for the DT and the entire set of patterns were used to train and build the DT. We used ID3 [8], a common algorithm for building DT. Classic ID3 stops growing the DT when each leaf contains items that should be classified to one class only. Since we had a very large set of patterns we stopped growing the tree as soon as the percentage of patterns of one of the groups (*larger* or *equal*) in the given tree node exceeded a predefined threshold th_1 (classic ID3 uses $th_1 = 100\%$). The exact value for th_1 was determined experimentally for the various domains. Similarly, once the number of patterns in a node was smaller than another threshold th_2 we stopped growing the DT. In nodes with mixed patterns we used the *majority* function to determine the class of this node.

4.2.2 Misclassification of the decision tree

Because of the early stopping condition, some of the patterns can be misclassified by the decision tree. There is no problem if a pattern of the *larger* group was misclassified as *equal*. In this case, we use the sum of the smaller PDBs. This value is admissible but might be smaller than the real value of the larger PDB. The other direction is more problematic. Here *equal* patterns were misclassified as *larger*. This will cause the ANN to have such patterns in its training set. But, recall that to preserve admissibility, all patterns with overestimated ANN values are stored in a hash table so admissibility is kept.

4.2.3 Experimental results

Line 5 in Table 1 shows the results of using the ANN+DT to compress the 6-6-3 additive PDB of the 15-puzzle. It shows that augmenting the basic ANN with the DT technique reduces the number

[5] In [3] *sparse* (multi dimensional array) mapping was used for the PDBs and thus the DIV method compressed cliques. Here, we used their more realistic *compact* mapping (a single dimensional array). The DIV method does not compress cliques here and its performance is worse than DIV for sparse mapping. See [3] for more details.

[6] In fact, as described earlier, we had two sets of smaller PDBS. We classified a pattern as *equal* if its heuristic was equal to the maximum of the sums of the two sets of smaller disjoint partitioning.

of nodes generated by roughly 20% (from 11,676,726 to 9,550,754) and reduces the memory requirements by 35% (from 1.3 to 0.84 Megabytes). The ANN now only handles the *larger* patterns. Not only it has fewer patterns to classify, these patterns have similar attributes. This allows it to be more accurate for the same amount of training. Consequently, the hash tables can be smaller because fewer patterns have their values overestimated by the ANN. Indeed the hash table percentage dropped from 8% to 4%.

4.3 Partitioning the patterns into groups (PART)

To properly train the ANN to have a reasonable error range it is necessary to feed it with the entire set of training instances at least 500 times. This can increase the total amount of training time especially if very large PDBs are used where data is saved on the disk.

To address this, we add another step before building the DT. The basic idea is to partition the patterns into smaller groups (for very large PDBs this can be done in the disk) and then (load each group into the memory and) build a separate DT+ANN system for each group. In order to classify these groups we use a k smaller heuristics (e.g., members of the heuristic vector). We call them the *pivot* heuristics. We then classify the patterns according to the values of the k pivot heuristics, For example assume that two members of the heuristic vector h_1 and h_2 are used. Pattern t with $h_1(t) = x$ and $h_2(t) = y$ will belong to the group labeled $\langle x, y\rangle$. Each such group contains patterns with similar attributes as they had similar values for the pivot heuristics. Each group will have a separate DT+ANN and the prediction will be more accurate due to similarities of the patterns inside each group. Another advantage is that for very large PDBs, which cannot be stored in memory, we can partition the large PDB into smaller groups which can fit in memory. Then, we build a DT+ANN for each group. Our full system of ANN+DT+Partitioning will be refereed to as ADP in the reminder of this paper.

Line 6 in Table 1 shows the results of using the full ADP system to compress the 6-6-3 PDBs. We used exactly the same heuristic vector as used for previous lines. Partitioning is done based on two heuristic values, each is the sum of 2- and 4-tile PDBs. Augmenting the ANN+DT system with the partitioning technique reduces the number of nodes generated by roughly 25% (from 9,550,754 to 7,285,207) and reduces the memory requirements by 40% (from 0.84 to 0.5 Megabytes). Compared to the original 6-6-3 PDB (line 1 in Table 1), ADP compression reduces the memory required by over 95%, while increasing the number of nodes generated by only 15%. It also significantly outperforms the $DIV2$ compression method of [3] in all aspects - nodes, time and memory.

4.4 The general framework for ADP

To summarize, the following preprocessing steps should be taken to build the full three-step ADP learning system:

- Create the original PDB.
- Create small PDBs for the heuristic vector and choose the pivot heuristics.
- Partition the patterns of the original PDB into small groups according to values of the pivot heuristics.
- Create a DT for each group of the partition and classifying patterns to *equal* or to *larger*.
- Train an ANN for patterns that were classified as *larger*.
- Test the ANN and build the hash table for overestimating patterns.

To obtain a heuristic value for a state s during the search we do the following:

- Extract the values of the heuristic vector for s and we find appropriate group according to the pivot heuristics.
- Traverse the relevant DT and see if it is a *larger* or *equal* node.
- If it is an *equal* node, add up the smaller PDB heuristics. If it is a *larger* node consult the relevant hash table and the relevant ANN and retrieve the heuristic value.

5 Experimental results

We now present additional experimental results for the full ADP system for the the 15- and 24-puzzles and for the TopSpin puzzle.

5.1 15-Puzzle

ADP was used to compress the 7- and 8-tile PDBs of the 7-8 additive PDB for the 15-puzzle (used in [4]). The two PDBs were compressed individually. The heuristic vector for each consisted of four values based on two $6-2$ additive PDBs, for the 8-tile PDB and on $6-1$ for the 7-tile PDB. These heuristics are also used as the pivot heuristics.

Heuristic	AvH	Nodes	Time	Mem	Hash
7-8	44.08	157,553	0.07	549	0
7-6-2	41.70	1,486,038	0.54	61	0
DIV 2	42.43	950,473	0.33	274	0
ADP (6-1)^2-(6-2)^2	43.03	307,332	0.21	46	2.9%
ADP (4-3)^2-(4-4)^2	41.96	899,516	0.57	16	2.2%

Table 2. ADP compression of the 7-8 additive PDB.

Table 2 presents the results in the same format as Table 1. The first row presents the results when using the normal, uncompressed additive 7-8 PDB. The second row is for an uncompressed 7-6-2 additive PDB. The next row is for the DIV 2 compressing of [3]. The next row is for ADP using heuristic vectors containing two $6-1$ additive PDBs for the 7-tile PDB and $6-2$ for the 8-tile PDB. The final row is for ADP using heuristic vectors containing two $4-3$ additive PDBs for the 7-tile PDB and two $4-4$ additive PDBs for the 8-tile PDB.

The last two lines show that varying the PDBs used in ADP's heuristic vector produces an interesting time-space tradeoff. However, both of these systems use less memory than the uncompressed 7-6-2 additive PDB and the DIV 2 compressing method and generate significantly fewer nodes. The ADP with $(6-1)^2-(6-2)^2$ was even faster in CPU time. Compared to the state-of-the-art uncompressed 7-8 PDB, this ADP reduces the memory required by over 90%, at a cost of less than doubling the number of nodes generated.

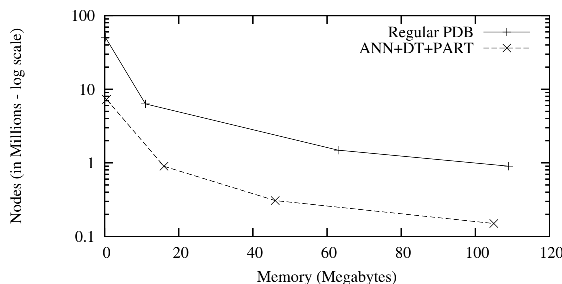

Figure 2. Nodes generated by both compression and regular systems.

Figure 2 brings together the data for uncompressed PDBs (solid line) and compressed PDBs using ADP (dashed line) from Tables 1 and 2 in order to compare the number of generated nodes as a function of the memory used. It also includes two data points not shown in those tables of 7-7-1 additive PDBs. This figure clearly

shows that for any given amount of memory it is far better to use a compressed PDB than a regular uncompressed PDB.

5.2 24-puzzle

The best existing heuristic for 24-puzzle uses a 6-6-6-6 additive PDB, and takes the maximum of the normal PDB lookup (r), its reflection about the main diagonal (r*), the dual lookup (d), and the reflection of the dual (d*) [5]. All values for regular lookup can be extracted from two 6-tile PDBs. For dual lookup, we need six additional PDBs [5]. ADP is applied to all these 6-tile PDBs. As in the 15-puzzle, the heuristic vector for the 6-tile PDB contains two additive 4-2 PDBs and they were also used for the partitioning step.

PDB Lookups	Nodes	Time	Mem	Hash
r,r*	43,454,810,045	15,861	244	0
r,r*,d,d*	13,549,943,868	8,441	972	0
r,r* (ADP)	69,527,696,072	31,843	4	1.6%
r,r*,d,d* (ADP)	19,781,408,283	15,971	37	1.9%

Table 3. Results for the 24-puzzle.

Table 3 shows the experimental results. The values are averages over the first 25 random instances used in [5]. Lines 1-2 are for the uncompressed PDBs, lines 3-4 are for the compressed PDBs. The first line in each group shows the results when only the regular lookup and its reflection are done. The second line in each group shows the results when the dual lookup and its reflection are done in addition to the regular lookups. By using two lookups, ADP decreased the size of PDB by a factor of 63 while increasing the number of nodes generated by only a factor of 1.6. For four lookups ADP decreased the size of PDB by a factor of 27 while increasing the number of generated nodes only by a factor of 1.45.

5.3 Top-spin

We also applied the ADP system on the N-TopSpin puzzle. A PDB of t tokens has actually N different ways of being used. A PDB of tokens [1 ... t] can also be used as a PDB of [2...t+1], [3...t+2], etc. with the appropriate mapping of tokens. Thus, a single PDB allows up to N different lookups. In separate experiments we applied ADP to a 9-token PDB and a 10-token PDB for the 17-TopSpin. The heuristic vector in each case contained 3 values corresponding to 3 different lookups in a PDB based on 7 tokens, for the 9-token PDB, and based on 8 tokens for the 10-token PDB. The partitioning of the 9- and 10-token PDBs useed all the PDBs from their heuristic vectors.

PDB	AvH	Nodes	Time	Mem	Hash
1 Lookup					
9	10.61	43,496,120	74.18	494	0
8	9.58	394,922,925	589.10	54	0
9 MOD	9.30	61,709,097	104.38	54	0
9 ADP	9.97	48,335,470	97.44	48	2.6%
2 Lookups					
9	10.96	664,966	1.62	494	0
8	10.01	5,777,064	11.29	54	0
9 MOD	9.68	6,489,343	14.71	54	0
9 ADP	10.20	1,475,642	4.29	48	2.6%
10	11.94	84,772	0.21	3,959	0
10 ADP	11.32	194,252	0.92	484	2.4%

Table 4. Results for (17,4)-TopSpin.

The experimental results are shown in Table 4, where each value is an average over a set of 100 random instances. Lines 1-4 show the results of solving 17-Topspin if just one lookup is made in the PDB, while rows 5-8 show the results if two lookups are made. The first two lines in each group show the results of using an uncompressed 9-token or 8-token PDB. The third line shows the results of the best compression technique used in [3] for 17-TopSpin, which compresses the table for the 9-token PDB using the MOD operator. The final row in each group is for our ADP compression technique. For both one and two lookups ADP clearly generates fewer nodes than the other techniques with similar amount of memory (the 8-token PDB and the 9-Token PDB compressed by the MOD operator. With two lookups it was even faster in CPU time. In fact, when two lookups are made the MOD method actually generates more nodes than an uncompressed PDB of the same size, the 8-token PDB.

The last two rows show the results of using regular and compressed 10-token PDB and with two lookups. ADP reduces the memory required by 87% while increasing the number of generated nodes by a factor of 2.3. The compressed version of the 10-token PDB requires slightly less memory than the uncompressed 9-token PDB but generates only 30% of the nodes and 56% of the time.

6 Summary and Conclusions

We presented a new technique that better utilizes memory by compressing PDBs with learning techniques and we applied it to different domains. A three step mechanism to construct the system was introduced but any subset of them can be separately used. A significant reduction in memory was achieved over the uncompressed PDB at a cost of a small increase in the search effort. Furthermore, our compressing idea usually outperforms previous compressing techniques in both memory and number of nodes and many times in CPU time as well. For a given amount of memory it is beneficial to use our compressing technique over uncompressed PDB of the same size.

An advantage of our system is that PDBs that are much larger than the available memory can be generated on disk and can be compressed to fit the memory. In fact, we used this method to compress 10-token PDB for 17-TopSpin.

Future work will continue these ideas as follows. First we would like to compress much larger PDBs as well as trying to solve larger versions of these puzzles. Second, other classifier techniques (like oblique tree and SVM [8]) might perform better that the ADP system. Finally, this approach can be applied in compressing PDBs in planning domains.

REFERENCES

[1] S. Edelkamp, 'Symbolic pattern databases in heuristic search planning', *AIPS*, 274–293, (2002).

[2] M. Ernandes and M. Gori, 'Likely-admissible and sub-symbolic heuristics.', in *ECAI*, pp. 613–617, (2004).

[3] A. Felner, R. Korf, R. Meshulam, and R. Holte, 'Compressed pattern databases', *JAIR*, **30**, 213–247, (2007).

[4] A. Felner, R. E. Korf, and S. Hanan, 'Additive pattern database heuristics', *JAIR*, **22**, 279–318, (2004).

[5] A. Felner, U. Zahavi, R. Holte, and J. Schaeffer, 'Dual lookups in pattern databases', in *Proc. IJCAI*, pp. 103–108, (2005).

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

[7] J. S. Judd, *Neural network design and the complexity of learning*, MIT Press, Cambridge, MA, USA, 1990.

[8] T. Mitchell, 'Machine learning and data mining', *Communications of the ACM*, **42**(11), 30–36, (1999).

[9] George Politowski, *On the construction of heuristic functions*, Ph.D. dissertation, University of California at Santa Cruz, 1986.

[10] S. Sarkar, P. Chakrabarti, and S. Ghose, 'A framework for learning in search-based systems', *IEEE Transactions on Knowledge and Data Engineering*, **10**(4), 563–575, (1998).