Multi-valued Pattern Databases

Carlos Linares López[1]

Abstract.

Pattern Databases were a major breakthrough in heuristic search by solving hard combinatorial problems various orders of magnitude faster than state-of-the-art techniques at that time. Since then, they have received a lot of attention. Moreover, pattern databases are also researched in conjunction with other domain-independent techniques for solving planning tasks. However, they are not the only technique for improving heuristic estimates. Although more modest, perimeter search can also lead to significant improvements in the number of generated nodes and overall running time. Therefore, whether they can be combined or not is a natural and interesting issue. While other researchers have recently proven that a joint application of both ideas (termed as *multiple goal*) leads to no progress at all, it is shown here that there are other alternatives for putting both techniques together —denoted here as *multi-valued*. This paper shows that *multi-valued* pattern databases can still improve the performance of standard (or *single-valued*) pattern databases in practice. It also examines how to enhance memory usage when comparing *multi-valued* pattern databases in contraposition to various *single-valued* standard pattern databases.

1 Introduction

Heuristics play a central role in problem-solving by guiding search algorithms towards the goal state from an arbitrary state, anywhere in the state space. Before the conception of *pattern databases* [1], heuristics were usually either handcrafted or directly derived by relaxing the original constraints of the problem at hand. In other words, pattern databases are an automatic mean for deriving heuristic functions which are usually far better informed than others. Thus leading to large improvements in the number of nodes generated and the overall running time.

However, since pattern databases can take large chunks of main memory, various alternatives have been explored to efficiently use the available memory. On one hand, it has been shown that pattern databases can be successfully compressed, at least, in some domains like the Towers of Hanoi [5]. Also, it has been shown that pattern databases can be *mapped* re-using the same symbol instead of using a one-to-one mapping [9] as originally suggested.

Although pattern databases can lead to further improvements by exploiting some domain-specific properties (e.g. reflections in the definition of the state or intrinsic characterizations of permutation state spaces [7]), they have been used also for solving planning tasks in conjunction with other domain-independent techniques [3] with very good results.

In contrast to pattern databases, *perimeter search* [2, 12] aims at improving an existing heuristic function, instead of automatically generating a new one. Although this technique has been usually employed for solving large sets of instances with respect to the same target, it could be broadly used when solving problems for distinct goal nodes.

Altogether, pattern databases can be used for automatically deriving heuristic functions, and perimeter search serves for improving its estimates. Hence, whether they can be combined or not is an interesting issue which has already been addressed [6]. However, the first results in this regard showed that perimeter search leads to no benefit at all. In this paper, a different technique for combining both ideas is discussed.

2 Background

This section succinctly reviews the main concepts underlying both perimeter search and pattern databases. The interested reader should refer to the cited papers for further information.

2.1 Perimeter Search

Perimeter search was independently and simultaneously introduced in the specialized bibliography by Giovanni Manzini [12] and John F. Dillenburg and Peter C. Nelson [2]. The key observation of these researchers is that the main problems in bidirectional search come from the fact that both searches progress simultaneously. They proposed, instead, to generate a set of nodes (known as *perimeter nodes*) whose descendants exceed a given threshold d (known as *perimeter depth*) around the target node, and only after it has been generated, to start a unidirectional search from the source state until a collision with a perimeter node is detected. From this point of view, perimeter search might be seen as a more simple form of bidirectional search. However, the most prominent feature of this contribution is that it provides a mean for automatically improving an existing heuristic function, $h(\cdot)$, since the unidirectional search from an arbitrary state, n uses the following, better informed, heuristic function h_d:

$$h_d(n,t) = \min_{m \in P_d} \{h(n,m) + h^*(m,t)\} \tag{1}$$

where P_d is the perimeter set comprising all nodes generated at depth d from the target, t, and $h^*(m,t)$ is the optimal cost of reaching the goal from the perimeter node m.

Although it can be argued that using perimeter search involves "*as many heuristic calculations as there are perimeter nodes*" [11], the truth is that this number decreases with the depth of the forward search [12].

2.2 Pattern Databases

In their original work [1], Joseph C. Culberson and Jonathan Schaeffer defined *patterns* as abstractions of the original state space where

[1] Planning and Learning Group, Universidad Carlos III de Madrid. Avda. de la Universidad, 30 - 28911 Leganés, Madrid (Spain) email: carlos.linares@uc3m.es

each constant appearing in the state space gets replaced by either a dedicated symbol or a special "don't care" symbol. The *granularity* of the abstraction is defined as the number of constants in the original state being replaced by the same symbol [8]. For example $\gamma = \langle 3, 3, 2, 1 \rangle$ denotes an abstraction where three constants are replaced by one symbol (say x_1); another three are replaced by a new symbol x_2; another two constants by a third symbol, x_3, and the last constant by a unique symbol, x_4. Although it has not been mentioned before in the related literature, it can be easily proven that the number of patterns generated with a given granularity γ is:

$$\prod_{i=1}^{|\gamma|} C_{\left(N-\sum_{j=1}^{i-1}\gamma_j\right),\gamma_i} = \prod_{i=1}^{|\gamma|} \binom{N - \sum_{j=1}^{i-1}\gamma_j}{\gamma_i} \tag{2}$$

where $C_{n,m}$ is the number of combinations of n elements choose m, and N is the total number of constants in the original state space, so that $N = \sum_i \gamma_i$. Thus, the previous granularity gives raise to:

$$C_{9,3} \times C_{9-(3),3} \times C_{9-(3+3),2} \times C_{9-(3+3+2),1} = 5,040$$

different entries.

Pattern databases are simply hash tables which store, for every pattern (or arrangement of symbols in the abstracted state), the minimum number of moves required to place the symbols in the abstracted state space in their goal location —also known as *goal pattern*. This value can be easily computed with a backwards brute-force breadth-first search from the goal pattern. So far, pattern databases are admissible heuristic functions. The index into the pattern database assigned to each pattern results from a ranking function which is (usually by far) the most expensive operation in searching with pattern databases[2]. Originally, all moves were counted in so that when comparing the values retrieved from different pattern databases (for a collection of different patterns), the only way for getting an admissible heuristic is just to take the MAX of all values. However, when the constants appearing in the original state space can be split into disjoint sets (as in the N-puzzle or the Towers of Hanoi, but not in the Rubik's cube or the TopSpin puzzle), a far better informed heuristic function can be built by computing the summation of all values [10]. This idea is known as disjoint, or just ADD pattern databases.

3 Combining Perimeter Search and Pattern Databases

As mentioned in the introduction, the main contribution of this work consists of discussing a different way than that previously proposed in [6] for putting together both perimeter search and pattern databases.

3.1 Mutiple Goal Pattern Databases

The first approach consists of addressing the combination as a multiple goal problem, i.e. a special case of heuristic search where the problem consists of hitting any of the perimeter nodes generated at depth d. A simple, yet beautiful, way for solving this sort of problems with the aid of pattern databases, consists of seeding the queue used in the backward breadth-first search with all the perimeter nodes [11]. This way, the pattern database will store a unique value per entry with the minimum distance to all perimeter nodes.

[2] *Ranking* consists of converting each item in a collection into a scalar.

The apparent advantage of this approach is that while standard pattern databases explore the abstracted search space around the goal node, the perimeter generation starts by considering the original state space up to a pre-defined perimeter depth.

Nevertheless, the same state can be mapped, in *different* pattern databases, to *different* entries which contain the minimum distance to *different* perimeter nodes, so that comparisons become more difficult. In other words, this idea is likely to produce very poor estimates by comparing the minimum distance to different perimeter nodes.[3] Besides, Ariel Felner and Nir Ofek [6] experimentally showed, and empirically proved, that this approach leads to no improvement at all, i. e., it generates the same number of nodes. Their explanation can be intuitively depicted as follows: the only expected benefit under this scheme is that patterns happening within the perimeter are now assigned better heuristic estimates, since patterns appearing beyond the perimeter set shall still get the same minimum distance. Comparing the number of patterns within the perimeter with all the plausible patterns gives a very small ratio in favour of this approach.

3.2 Multi-valued Pattern Databases

Instead, it is suggested herein to store separately the distance to each perimeter node in the pattern database, as shown in figure 1. Thereby, comparisons with respect to the same perimeter node become now feasible, leading to a better informed heuristic function as discussed in section 2.1. This is, indeed, the most natural way to implement perimeter search. Since every entry contains a vector of values instead of a scalar, this technique is denoted as *multi-valued* pattern databases in contraposition to standard *single-value* pattern databases which consist of a unique value per entry.

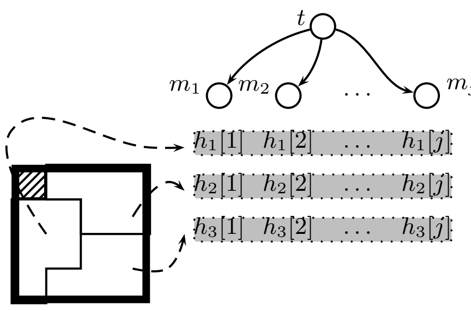
t
m_1
m_2
$\dots$
m_j
$h_1[1]$ $h_1[2]$ $\dots$ $h_1[j]$
$h_2[1]$ $h_2[2]$ $\dots$ $h_2[j]$
$h_3[1]$ $h_3[2]$ $\dots$ $h_3[j]$

Figure 1. Seeding a different queue with every perimeter node

At first glance, it might seem that this approach wastes a lot of space in main memory. However, this is not the case at all in the vast majority of cases. Consider, for example, the 15-Puzzle and a *single-valued* pattern database consisting of 7 different symbols, i.e. $\langle 1, 1, 1, 1, 1, 1, 1, 9 \rangle$. According to equation (2), this yields 57,657,600 different entries. What is the next bigger pattern database that can be built?

- One option consists of augmenting the original pattern database with an additional symbol, this is, taking 8 different constants,

[3] Indeed, though not explicitly mentioned in [6], the termination condition might become more difficult now, since it is not strictly true that when various pattern databases return zero, a collision with a perimeter node has been detected, since maybe they are all referring to different nodes!

whose granularity is $\langle 1,1,1,1,1,1,1,1,8\rangle$. This new, bigger, pattern database consists of 518,918,400 entries and is 9 times bigger than the original one.

- Another option consists of mapping an additional constant in the original state space to some other symbol currently used. This case is represented with granularity $\langle 1,1,1,1,1,1,2,8\rangle$ and originates 259,459,200 entries, 4.5 times more than the original pattern database.

However, the number of perimeter nodes generated in the 15-Puzzle at depth $d = 1$ and 2 is $|P_d| = 2$ and 4, respectively. This means that the resulting *multi-valued* pattern databases are smaller than the *single-valued* pattern databases created in both cases.

Another consideration tightly related to the size of the resulting pattern databases is the number of ranking operations performed in each case. While the number of nodes to consider simultaneously in *multi-valued* pattern databases impose an overhead, they are all retrieved in a row, i.e. with a single ranking operation. This is true because the distances to each perimeter node are stored in contiguous locations in memory. However, if different *single-valued* pattern databases are going to be employed (which take altogether the same space than a *multi-valued* pattern database), each value shall be retrieved separately so that various ranking operations shall be performed. Since ranking is the most expensive operation in pattern databases, this overhead shall be taken into account as well.

3.3 Results

Although ADD pattern databases are known to provide more accurate heuristic values, it is not always possible to apply them. Therefore, experiments have been conducted with both ADD and MAX pattern databases.

In both cases, the perimeter is generated using a brute-force depth-first search algorithm from the goal which generates all nodes whose descendants have a cost that exceeds the specified perimeter depth. Once the perimeter set is generated, different queues are seeded with each perimeter node and a backward breadth-first search is issued with everyone for a given pattern specification. As a result, *multi-valued* pattern databases (which are as many times bigger than a *single-valued* pattern database as perimeter nodes were found) are generated.

For the ease of comparison, the same mapping functions have been programmed. Since *sparse mapping* incurs in prohibitive wastes of space for some cases, a *compact mapping* has been chosen[4].

Because IDA* explores the state space in a depth-first fashion, an incremental implementation of the Myrvold and Ruskey ranking algorithm [13] has been developed. The current implementation runs about 20%–30% faster and has no additional memory requirements. Unfortunately, due to space constraints no further details regarding this algorithm are provided. It is worth mentioning that *single-valued* pattern databases are expected to be more sensitive to this improvement than *multi-valued* pattern databases. The reason is that *multi-valued* pattern databases, being more informed than *single-valued* pattern databases (according to section 2.1) will generate and rank less nodes.

3.3.1 *Multi-valued* ADD *pattern databases*

The domains chosen for experimenting with *multi-valued* ADD pattern databases are the 15-Puzzle and the 24-Puzzle. In all cases, pattern databases are "*blank-preserving*" [8], i. e., the blank tile is always mapped to a unique symbol, instead of "*blank-increasing*" — which consists of mapping the blank tile to the same symbol used by other tiles, such as the "don't care" symbol. Also, reflections about the main diagonal are computed for *single-valued* pattern databases only if the regular lookup did not exceed the current threshold. Since no domain dependent feature is exploited for *multi-valued* pattern databases, results with reflections are provided only for the sake of completeness.

Table 1 shows the mean time elapsed (in seconds) and the total number of generated nodes for solving the Korf's test suite, which consists of 100 problems, when using *single-valued* and *multi-valued* pattern databases. In the experiments, six different arrangements of pattern databases have been used, where each pattern consists of 5 pattern tiles —pattern database #6 is the same suggested in [4]. In all tables, *sPDB* denotes *single-valued* pattern databases; *mPDB*$_i$ stands for *multiple-valued* pattern databases generated with perimeter depth $d = i$ and, finally, *rPDB* stands for the same pattern databases as in *sPDB* but taking advantage of the reflections about the main diagonal.

5-5-5	#1	#2	#3	#4	#5	#6
sPDB (0.0014Gb)	1.28 0.851	0.56 0.324	2.17 1.561	0.79 0.456	2.32 1.634	0.47 0.254
mPDB$_2$ (0.0058Gb)	0.49 0.188	0.56 0.232	0.92 0.417	0.81 0.364	1.18 0.570	0.64 0.253
rPDB (0.0014Gb)	1.01 0.372	0.38 0.111	1.48 0.581	0.49 0.140	1.49 0.585	0.44 0.134
6-6-3	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**
sPDB (0.0107Gb)	0.83 0.509	0.98 0.576	0.42 0.187	0.77 0.452	1.77 1.183	0.39 0.181
mPDB$_2$ (0.0429Gb)	0.37 0.113	0.45 0.144	0.30 0.086	0.33 0.108	1.00 0.434	0.50 0.181
rPDB (0.0107Gb)	0.78 0.248	0.63 0.186	0.36 0.092	0.44 0.119	1.79 0.638	0.24 0.056

Table 1. Experimental results in the 15-Puzzle with 5-5-5 and 6-6-3 PDBs: each cell shows the mean run-time in seconds (above) and total number of generated nodes in thousands of millions, 10^9 (below)

Table 1 shows also the same statistics for another six different arrangements of 6-6-3 pattern databases. Pattern database #6 is the one suggested in [4]. Next, table 2 depicts the same statistics for four different arrangements of 7-8 pattern databases. In this case, pattern database #1 is the one widely suggested in the specialized bibliography and also cited in [4].

	#1	#2	#3	#4
sPDB (0.5369Gb)	0.0368 13.721	0.1338 60.347	0.1374 54.323	0.1687 78.370
mPDB$_2$ (2.1479Gb)	0.0355 10.262	0.0434 15.374	0.0435 15.502	0.0580 18.349
rPDB (0.5369Gb)	0.0088 3.832	0.0993 21.646	0.0846 19.379	0.0939 23.594

Table 2. Experimental results in the 15-Puzzle with 7-8 PDBs: mean run-time in seconds (above) and total number of generated nodes in millions, 10^6 (below)

Table 3 shows the same statistics in the 24-Puzzle using two different arrangements of 6-6-6-6 pattern databases at depth 3. Pattern database #1 is the usual reference in this domain, as suggested in [4]. The test set employed consists of the 25 easiest instances of the test

[4] For a thorough discussion on the topic, see [4], section 4.2, page 289.

	#1	#2
sPDB (0.4750Gb)	10622.39 3.08	24746.44 7.02
mPDB_3 (4.7501Gb)	6162.81 1.00	12227.45 1.93
rPDB (0.4750Gb)	2390.09 0.41	10170.26 1.72

Table 3. Experimental results in the 24-Puzzle with 6-6-6-6 PDBs: mean run-time in seconds (above) and total number of generated nodes in millions of millions, 10^{12} (below)

suite detailed in [10], with solution lengths ranging from 81 to 106 moves.

When comparing the performance of various *single-valued* pattern databases versus their *multi-valued* counterparts, it turns out that the latter usually outperforms the former, more remarkably in the 7-8 and 6-6-6-6 cases. But this is not always true —see, for example PDB #6 in the 6-6-3. However, when comparing all running-times, the pattern database which resulted in faster performance is always the *multi-valued* pattern database (for example, in the 6-6-3 case, the fastest algorithm uses *multi-valued* pattern databases arranged as in #3), but in the 5-5-5 case. The fact that for some arrangements, *multi-valued* pattern databases do not outperform their *single-valued* counterpart but others do it, can be explained as an effect of the diversity induced by the perimeter nodes. It has been observed that for some arrangements of pattern databases, the blank tile only reaches a few patterns when computing the perimeter nodes. The more pattern databases are affected, the better the heuristic. For example, in the 7-8 PDB #1 of the 15-Puzzle[5], allowing the blank to move twice affects both pattern databases. Thus, the resulting *multi-valued* pattern database outperformed its *single-valued* counterpart, even though the latter is very accurate for solving this problem. Correspondingly, when computing the *multi-valued* pattern database of 5-5-5 #6, only one PDB out of three gets updated, thus not leading to any improvement on either the number of nodes generated or the running time.

3.3.2 *Multi-valued* MAX *pattern databases*

The domain chosen for these experiments is the (N, K)-TopSpin. Max'ing is far less efficient than taking the summation of a few values from different disjoint pattern databases. Thus, the sizes of the instances considered here are smaller than the ones shown in the previous paragraph. The number of pattern databases and the number of tiles they contain in each case is clearly identified in the tables. For example, 6-6 stands for two pattern databases with 6 tiles each. Besides, they always consist of contiguous locations arranged in such a way that the pattern databases are all equidistant, thus minimizing the overlapping among them. In all the subsequent experiments, the test suites employed consisted of 100 solvable instances generated with the random application of a number of operators between 100 and 500.

Table 4 shows the results in the $(9, 2)$-TopSpin. This puzzle can be solved so fast that in most cases the time spent falls below 0.00 seconds. The number of perimeter nodes generated at depth $d = 1$ and 2 is $|P_d| = 3$ and 6, respectively, so that mPDB_1 and mPDB_2 are 3 and 6 times larger than the size of the corresponding sPDB, shown below every arrangement. As it can be seen, the overhead imposed by perimeter search clearly pays-off for the reduction on the number of nodes generated.

[5] In this case, the 15-Puzzle is split into two halves: one above the other. The one below contains 8 pattern tiles whereas the one over it contains 7, because the blank tile is ommitted.

	4-4-4 (0.0086Mb)	5-5-5 (0.0432Mb)	6-6-6 (0.1730Mb)
sPDB	0.01 5.952	≤ 0.00 0.458	≤ 0.00 0.104
mPDB_1	0.01 3.964	≤ 0.00 0.347	≤ 0.00 0.077
mPDB_2	≤ 0.00 3.044	≤ 0.00 0.263	≤ 0.00 0.060

Table 4. Experimental results in the $(9, 2)$-TopSpin: mean run-time in seconds (above) and total number of generated nodes in tenths of millions, 10^5 (below)

Table 5 summarizes the results for both the $(12, 2)$-TopSpin and the $(15, 2)$-TopSpin. As it can be seen, *multi-valued* pattern databases solved the problems faster and generating less nodes in all cases, with no exception.

	$(12, 2)$-TopSpin		$(15, 2)$-TopSpin
	6-6 (1.2689Mb)	8-8 (38.0676Mb)	7-7-7-7-7 (154.6497Mb)
sPDB	9.94 1.813	0.21 0.021	22.48 2.776
mPDB_1	6.54 1.066	0.16 0.013	20.16 2.114
mPDB_2	6.07 0.795	0.11 0.008	16.95 1.572

Table 5. Experimental results in the $(12, 2)$-TopSpin and the $(15, 2)$-TopSpin: mean run-time in seconds (above) and total number of generated nodes in thousands of millions, 10^9 (below)

4 Compressing Multi-valued Pattern Databases

From equation (2) it becomes clear that the number of patterns grows rapidly for any granularity. Thus, techniques have been developed for efficiently compressing pattern databases both in a lossy and lossless way [5]. In this section, some preliminary ideas for compressing *multi-valued* pattern databases as well are discussed. It should be highlighted that the techniques discussed herein are not incompatible with those introduced in [5].

In spite of the discussions in section 3.2, the truth is that disjoint (or ADD) *multi-valued* pattern databases take even less space than what it might seem. Consider the 7-8 PDB #1 for the 15-Puzzle generated with perimeter depth $d = 1$ —see fotnote 5. It is easy to realize that in the two perimeter nodes generated so far, the inferior half (i. e., the pattern database with 8 tiles) looks exactly the same than in the goal state. Since ADD pattern databases do count all moves of the blank tile, the values stored in the inferior *multi-valued* pattern database are likely to be the same. Thus, it is only necessary to store two values per entry in the superior pattern database, but only one in the inferior database. This way, the resulting *multi-valued* pattern databases take twice the space of the smaller *single-valued* database (the one with 7 tiles) but only once the space of the inferior, larger, *single-valued* database. This stands for a marginal increment in the size of 10%. Even considering larger perimeter depths (say $d = 2$), it is still possible to apply other compression schemes to *multi-valued* pattern databases as discussed below.

This is not true, however, for MAX pattern databases because in this case only moves of the pattern tiles are taken into account. Nevertheless, it is still possible to compress the resulting *multi-valued* pattern database relating statistically the distribution of values to each perimeter node with the distance to the first perimeter node.

Let $\delta_i(j)$ denote the difference $h_i(j)-h_i(1)$ where $h_i(j)$ is the j-th component of the vector in the i-th entry of a *multi-valued* pattern database. In other words, $\delta_i(j)$ is the difference of the distance to the j-th perimeter node and the first perimeter node from pattern i. This way, it is possible to compute the vector of differences $\delta_i(\cdot)$ for every entry, i, in a given *multi-valued* pattern database. Also, it is assumed that P perimeter nodes have been generated.

Now, there are two different ways to compress data in a loosy way without sacrificing admissibility:

Traversal compression consists of forcing all $h_i(j)$ values *from the same entry* i to be equal to the minimum of them all, so that each component j takes a new value $h'_i(j)$ computed as follows: $h'_i(j) = h_i(1) + \min_{j=2}^{P}\{\delta_i(j)\}, 2 \leq j \leq P$. The expected loss in the accuracy of the resulting heuristic values due to the traversal compression, $\mathcal{L}_t$, can be computed as:

$$\mathcal{L}_t(\delta_i(\cdot)) = \sum_{j=2}^{P}\left(h_i(j) - h'_i(j)\right)p(\delta_i)$$

where $p(\delta_i)$ stands for the probability of occurrence of the vector of differences δ_i. Note that the same vector of differences δ_i can happen in an arbitrary number of entries in the *multi-valued* pattern database other than the i-th entry.

Applying repeatedly this compression scheme, the resulting pattern database will be exactly the same than the one generated under the *multiple goal* approach in section 3.1.

Longitudinal compression merges two different entries, u and v, by forcing their δ vectors, $\delta_u(\cdot)$ and $\delta_v(\cdot)$, to be the same so that $h_u(i)$ and $h_v(i)$ take new values, $h'_u(i)$ and $h'_v(i)$, according to: $h'_u(i) = h_u(1) + \min\{\delta_u(i), \delta_v(i)\}$ and, similarly for h'_v. As in the previous case, it is possible to compute the expected loss in the accuracy of the heuristic values that result after a longitudinal compression, $\mathcal{L}_l$, as follows:

$$\mathcal{L}_l(\delta_u(\cdot), \delta_v(\cdot)) = \sum_{j=2}^{P}(h_u(j) - h'_u(j))p(\delta_u) + (h_v(j) - h'_v(j))p(\delta_v)$$

Since the preceding expressions allow the measurement of the loss in the accuracy of the heuristic function, they serve for compressing any *multi-valued* pattern database to any desired ratio of compression degree versus loss of accuracy. In particular, for any upper bound on the average loss of the heuristic function, U, an algorithm for efficiently compressing a *multi-valued* pattern database proceeds in the following fashion: if the average loss is still below U, compute the expected loss of all the traversal compressions, and also the expected loss of all the longitudinal compressions for each pair of entries in the pattern database. Next, pick the compression with the minimum expected loss and update the pattern database. Proceeding in this manner, the number of differences, $\delta(\cdot)$, will be monotonically decreasing at each step. If there are n different vectors of differences when the expected loss reaches the upper bound, U, code each entry in the pattern database with one of the indexes in the range $[1, \log_2 n]$, so that $log_2 n$ bits are used instead of 8, which is the usual choice. In other words, instead of storing a vector of heuristic estimations to each perimeter node in every entry of the *multi-valued* pattern database, an index to a small number of $\delta(\cdot)$ vectors is attached. Then, when solving a problem, retrieve the index from the pattern database and apply its $\delta(\cdot)$ vector to get the heuristic estimations to all the perimeter nodes. Preliminary experiments in the (N, K)-TopSpin suggest that it is feasible to significantly compress *multi-valued* pattern databases and still running faster than various *single-valued* pattern databases generating far less nodes.

5 Summary

Although it might be contrary to intuition, storing various values per entry in a pattern database can outperform the standard, *single-valued* pattern databases, either ADD or MAX. Furthermore, these databases can be compressed with the techniques outlined in the last section which are not incompatible with existing techniques for compressing *single-valued* pattern databases.

Acknowledgements

This work has been partially supported by the Spanish MEC project TIN2005-08945-C06-05 and UC3M-CAM project CCG06-UC3M/TIC-0831.

REFERENCES

[1] Joseph C. Culberson and Jonathan Schaeffer, ‘Pattern databases’, *Computational Intelligence*, **14**(3), 318–334, (1998).

[2] John F. Dillenburg and Peter C. Nelson, ‘Perimeter search’, *Artificial Intelligence*, **65**, 165–178, (1994).

[3] Stefan Edelkamp, ‘External symbolic heuristic search with pattern databases’, in *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05)*, pp. 51–60, Monterey, California, United States, (June 2005).

[4] Ariel Felner, Richard E. Korf, and Sarit Hanan, ‘Additive pattern database heuristics’, *Journal of Artificial Intelligence Research*, **22**, 279–318, (November 2004).

[5] Ariel Felner, Richard E. Korf, Ram Meshulam, and Robert Holte, ‘Compressed pattern databases’, *Journal of Artificial Intelligence Research*, **30**, 213–247, (October 2007).

[6] Ariel Felner and Nir Ofek, ‘Combining perimeter search and pattern database abstractions’, in *Proceedings of the Seventh Symposium on Abstraction, Reformulation and Approximation (SARA-07)*, pp. 155–168, Whistler, Canada, (July 2007).

[7] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte, ‘Dual lookups in pattern databases’, in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pp. 103–108, Edinburgh, Scotland, (July 2005).

[8] Robert Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy, ‘Multiple pattern databases’, in *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pp. 122–131, Whistler, British Columbia, Canada, (June 2004).

[9] Robert C. Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy, ‘Maximizing over multiple pattern databases speeds up heuristic search’, *Artificial Intelligence*, **170**(16–17), 1123–1136, (November 2006).

[10] Richard E. Korf and Ariel Felner, ‘Disjoint pattern database heuristics’, *Artificial Intelligence*, **134**(1–2), 9–22, (2002).

[11] Richard E. Korf and Ariel Felner, ‘Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem’, in *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 2324–2329, Hyderabad, India, (January 2007).

[12] Giovanni Manzini, ‘BIDA*: an improved perimeter search algorithm’, *Artificial Intelligence*, **75**, 347–360, (1995).

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