# Maximizing over Multiple Pattern Databases Speeds up Heuristic Search

Robert C. Holte [a] Ariel Felner [b] Jack Newton [a] Ram Meshulam [c] David Furcy [d]

[a] *University of Alberta, Computing Science Department, Edmonton, Alberta, T6G 2E8, Canada email: {holte,newton}@cs.ualberta.ca*

[b] *Department of Information Systems Engineering, Bar-Gurion University of the Negev, Beer-Sheva, Israel 85105 email: felner@bgu.ac.il*

[c] *Computer Science Department, Bar-Ilan University, Ramat-Gan, Israel 92500 email: meshulr1@cs.biu.ac.il*

[d] *Computer Science Department, University of Wisconsin Oshkosh, 800 Algoma Boulevard, Oshkosh, WI 54901-8643, USA email: furcyd@uwosh.edu*

**Abstract**

A pattern database (PDB) is a heuristic function stored as a lookup table. This paper considers how best to use a fixed amount ($m$ units) of memory for storing pattern databases. In particular, we examine whether using $n$ pattern databases of size $m/n$ instead of one pattern database of size $m$ improves search performance. In all the state spaces considered, the use of multiple smaller pattern databases reduces the number of nodes generated by IDA*. The paper provides an explanation for this phenomenon based on the distribution of heuristic values that occur during search.

## 1 Introduction and overview

### *1.1 Problem Statement*

Heuristic search algorithms such as A* [9] and IDA* [16] find optimal solutions to state space search problems. They visit states guided by the cost function $f(s) = g(s) + h(s)$, where $g(s)$ is the actual distance from the initial state to the current state $s$ and $h(s)$ is a heuristic function estimating the cost from $s$ to a goal state.

Pattern databases [2] provide a general method for defining a heuristic function, which is stored as a lookup table. They are the key breakthrough enabling various combinatorial puzzles such as Rubik's Cube [17], the sliding tile puzzles [6,19] and the 4-peg Towers of Hanoi problem [6,7] to be solved optimally. They have also led to significant improvements in the state of the art in heuristic-guided planning [4], model checking [5,22], and sequencing ([10], and Chapter 5 of [11]).

The method used to define pattern databases (see Section 2) is simple and has three important properties. First, it guarantees that the resulting heuristic functions are admissible and consistent [15]. Second, it gives precise control over the size of the pattern databases, so that they can be tailored to fit in the amount of memory available. Third, it makes it easy to define numerous different pattern databases for the same search space. The significance of the third property is that it enables multiple pattern databases to be used in combination. Two (or more) heuristics, $h_1$ and $h_2$, can be combined to form a new heuristic by taking their maximum, that is, by defining $h_{max}(s) = max(h_1(s), h_2(s))$. We refer to this as "maximizing over" the two heuristics. Heuristic $h_{max}$ is guaranteed to be admissible (or consistent) if $h_1$ and $h_2$ are. In special circumstances (see Section 4), it is possible to define a set of pattern databases whose values can be added and still be admissible [6,19].

With the large memory capacity that is available on today's computers, the general question arises of how to make the best use of available memory to speed up search? In this paper we address the memory utilization question by considering whether it is better to use all the available memory for one pattern database or to divide the memory among several smaller pattern databases. Many successful applications of pattern databases have used multiple pattern databases. For example, the heuristic function used to solve Rubik's Cube in [17] is defined as the maximum of three pattern database heuristics. However, the study presented in this paper is the first[1] to systematically examine whether using multiple smaller pattern databases is superior to using one large pattern database, and to systematically compare how performance varies as the number of pattern databases is changed.

## *1.2 Contributions and Outline of the Paper*

Our first set of experiments compares maximizing over $n$ pattern databases of size $m/n$ for various values of $n$ and fixed total size of the pattern databases, $m$. These experiments show that large and small values of $n$ are suboptimal – there is an intermediate value of $n$ that reduces the number of nodes generated by as much as an order of magnitude over $n = 1$ (one pattern database of size $m$). Our second set of experiments investigates maximizing over additive groups of pattern databases. The performance of one additive group of maximum-size pattern databases is compared to the performance obtained by maximizing over a number of different additive groups of smaller pattern databases. Here again the use of several, smaller

[1] An early version of this paper appeared in [14].

pattern databases reduces the number of nodes generated. A third experiment, reported in detail in [13] and summarized here, demonstrates the same phenomenon in the framework of hierarchical heuristic search.

The phenomenon exhibited in these experiments, namely that the number of nodes generated during search can be reduced by using several, smaller pattern databases instead of one maximum-size pattern database, is the paper's main contribution.

An equally important contribution is the explanation of this phenomenon, which is based on the distribution of heuristic values that occur during search. In particular, we demonstrate that if heuristics $h_1$ and $h_2$ have approximately equal mean values, but $h_1$ is more concentrated around its mean, then $h_1$ is expected to outperform $h_2$.

A final contribution is the observation that IDA*'s performance can actually be degraded by using a "better" heuristic. We give an example that arose in our experiments in which heuristics $h_1$ and $h_2$ are consistent and $h_1(s) \geq h_2(s)$ for all states, but IDA* expands more nodes using $h_1$ than it expands using $h_2$. The underlying cause of this behavior is explained.

The paper is organized as follows. Section 2 defines pattern databases and provides motivation for maximizing over multiple heuristics. Sections 3 and 4 describe experiments with maximizing over multiple pattern databases on various state spaces. Section 6 explains the phenomena that were obtained in the experiments. Section 7 describes an unusual circumstance in which the methods described in this paper fail. Finally, Section 8 gives our conclusions.

## 2 Pattern Databases

### *2.1 Basic Concepts*

The basic concepts of pattern databases are best illustrated with the classic AI testbed, the sliding-tile puzzle. Three versions of this puzzle are the $3 \times 3$ 8-puzzle, the $4 \times 4$ 15-puzzle and the $5 \times 5$ 24-puzzle. They consist 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 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 state, the 15-puzzle contains about $10^{13}$ reachable states, and the 24-puzzle contains almost $10^{25}$ states. These puzzles in their goal states are shown in Figure 1.

The "domain" of a search space is the set of constants used in representing states.

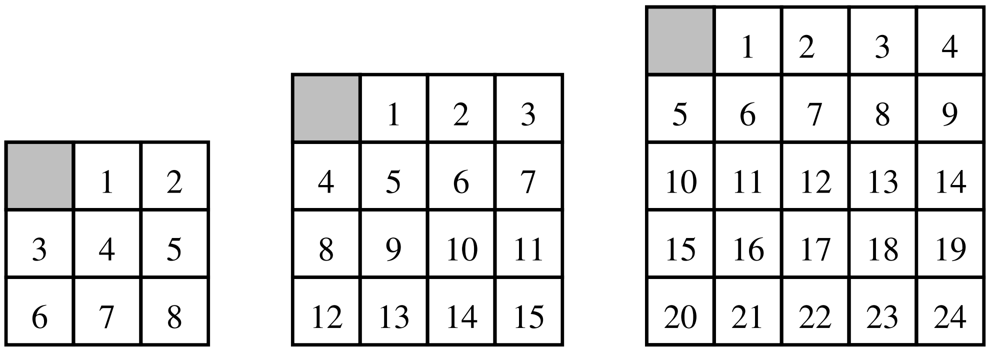


Fig. 1. The 8-, 15- and 24-puzzle goal states

For example, the domain of the 8-puzzle might consist of constants $1 \ldots 8$ representing the tiles and a constant, *blank*, representing the blank. [2]

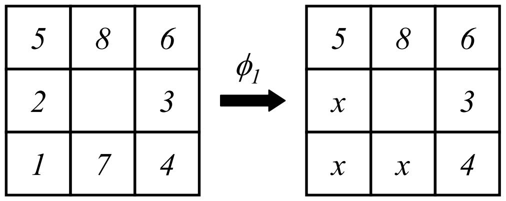


Fig. 2. Abstracting a state makes a pattern.

In the original work on pattern databases [3] a *pattern* is defined to be a state with one or more of the constants replaced by a special "don't care" symbol, $x$. For example, if tiles $1, 2$, and $7$ were replaced by $x$, the 8-puzzle state in the left part of Figure 2 would be mapped to the pattern shown in the right part of Figure 2. The *goal pattern* is the pattern created by making the same substitution in the goal state. [3]

The rule stating which constants to replace by $x$ can be viewed as a mapping from the original domain to a smaller domain which contains some (or none) of the original constants and the special constant $x$. Such a mapping is called a "domain abstraction." Row $\phi_1$ of Table 1 shows the domain abstraction just described, which maps constants $1, 2$, and $7$ to $x$ and leaves the other constants unchanged. A simple but useful generalization of the original notion of "pattern" is to use several distinct "don't care" symbols. For example, in addition to mapping tiles $1, 2$, and $7$ to $x$, one might also map tiles $3$ and $4$ to $y$, and tiles $6$ and $8$ to $z$. Row $\phi_2$ in Table 1 shows this domain abstraction. Every different way of creating a row in Table 1 with 8 or fewer constants defines a domain abstraction. These can be easily enumerated to create the entire set of possible domain abstractions for a domain.

[2] The definitions given for "domain" and "domain abstraction" assume that all state variables range over the same set of values and that it is desirable to apply the same abstraction to all state variables. These assumptions suffice for the studies in this and previous pattern database papers, but richer definitions might be needed in other circumstances.

[3] We assume that a single, fully-specified goal state is given.

| original | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $blank$ |
|---|---|---|---|---|---|---|---|---|---|
| $\phi_1$ | $x$ | $x$ | 3 | 4 | 5 | 6 | $x$ | 8 | $blank$ |
| $\phi_2$ | $x$ | $x$ | $y$ | $y$ | 5 | $z$ | $x$ | $z$ | $blank$ |

Table 1
Examples of 8-puzzle domain abstractions.

The patterns are connected to one another so as to preserve the connectivity in the original state space. [4] If an operator transforms state $s_1$ into state $s_2$ then there must be a connection from $\phi(s_1)$, the pattern corresponding to $s_1$, to $\phi(s_2)$, the pattern corresponding to $s_2$. The resulting set of patterns and their connections constitute the *pattern space*. The pattern space is an abstraction of the original state space in the sense that the distance between two states in the original space is greater than or equal to the distance between the corresponding patterns. Therefore, the distance between the two patterns can be used as an admissible heuristic for the distance between the two states in the original space.

A pattern database is a lookup table with one entry for each pattern in the pattern space. It is indexed by an individual pattern, and the value stored in the pattern database for pattern $p$ is the distance in the pattern space from $p$ to the goal pattern. A pattern database is most efficiently constructed by running a breadth-first search backwards from the goal pattern until the whole pattern space is spanned. The time to construct a pattern database can be substantially greater than the time to solve a single search problem instance with the pattern database (hours compared to fractions of a second), but can be amortized if there are many problem instances with the same goal to be solved. If there is only a small number of instances with the same goal to be solved, it is possible to build, on demand, the portion of the pattern database that is needed [13,15].

Given a state, $s$, in the original space, and a pattern database defined by the abstraction $\phi$, $h(s)$ is computed as follows. First, the pattern $\phi(s)$ is computed. This pattern is then used as an index into the pattern database. The entry in the pattern database for $\phi(s)$ is looked up and used as the heuristic value, $h(s)$.

## *2.2 Maximizing over Multiple Pattern Databases*

### *2.2.1 Motivation*

To motivate the use of multiple pattern databases, consider the two abstractions for the 15-puzzle, $\phi_7$ and $\phi_8$, defined in Table 2. The pattern database based on $\phi_7$ will be quite accurate in states where the tiles $8-15$ (the tiles it maps to $x$) are in their goal locations. On the other hand, it will be grossly inaccurate whenever

[4] Like most previous pattern database studies, we assume all operators have the same cost.

these tiles are permuted among themselves and tiles $1 - 7$ and the blank are in their goal locations. All such states map to the $\phi_7$ goal pattern and therefore have a heuristic value of 0 according to the $\phi_7$ pattern database. Domain abstraction $\phi_8$ has analogous strengths and weaknesses.

Because $\phi_7$ and $\phi_8$ are based on complementary set of tiles, the states where one of these pattern databases returns a very poor value might be precisely the states where the other pattern database is fairly accurate. Consulting both pattern databases for any given state and taking the maximum of the values they return is therefore likely to produce much more accurate values than either pattern database used alone. For example, when $\phi_7$ and $\phi_8$ are used together no state other than the goal will have a heuristic value of zero.

| original | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $blank$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi_7$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | $x$ | $x$ | $x$ | $x$ | $x$ | $x$ | $x$ | $x$ | $blank$ |
| $\phi_8$ | $x$ | $x$ | $x$ | $x$ | $x$ | $x$ | $x$ | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $blank$ |

Table 2
7-tile and 8-tile abstractions of the 15-puzzle.

### *2.2.2 Early stopping*

Given a set of pattern databases, $h(s)$ is computed by looking up a value in each pattern database and taking the maximum. In order to reduce the overhead of performing a number of lookups the loop over the pattern databases that computes the maximum can terminate as soon it encounters a heuristic value that makes $f(s) = g(s) + h(s)$ exceed the current IDA* threshold. State $s$ can be pruned immediately without the need to finish computing the maximum. To make this early stopping even more effective, the pattern database that had the largest value when computing $h(parent(s))$ is consulted first when computing $h(s)$.

Early stopping can potentially increase the number of nodes generated by IDA* compared to plain maximization. For example, suppose that the current threshold is 10, and a large tree has been searched without finding the goal. IDA* will increase the threshold to the smallest $f$-value exceeding 10 among the nodes generated in this iteration. For simplicity, suppose that all $f$-values that exceeded 10 in this iteration were 12 or greater with the possible exception of a single state, $c$. There are two pattern databases, $PDB_1$, which has the value 1 for $c$, and $PDB_2$, which has the value 2 for $c$. If we take the plain maximum over the two pattern databases the maximum will be 2 and the next threshold will be $f(c) = g(c) + h(c) = 12$. However, early stopping would stop as soon the value 1 was retrieved from $PDB_1$, because it makes $f(c) = g(c) + 1 = 11$ exceed the current threshold 10. Therefore, the next threshold becomes 11. This constitutes an iteration that was not present in the plain version, and it is entirely a waste of time because no new nodes will be expanded, not even $c$ itself. The only effect of this iteration is to do the full maximum computation for $h(c)$, which raises the depth bound to 12 once the useless iteration finishes. Thus, in certain circumstances early stopping can increase the number of iterations, and therefore potentially increase the number of nodes generated by IDA*. In some cases, however, the sequence of IDA* thresholds is fully determined by the state space and the abstraction used, and early stopping cannot lead to extra iterations. For example, the IDA* thresholds of the sliding tile puzzles increase by exactly two in the standard setting [6].

## 3 Experimental Results - Rubik's Cube

### *3.1 Overview*

In this section we compare the performance of IDA* with heuristics defined using $n$ pattern databases of size $m/n$ for various values of $n$ and fixed $m$, where $m$ is the size of the memory, measured as the number of pattern database entries. Our testbed in these experiments is Rubik's Cube, a standard benchmark domain for heuristic search that was first solved optimally using general-purpose techniques by Korf [17]. Shown in Figure 3, Rubik's Cube is made up of 20 movable sub-cubes, called cubies: eight *corner cubies*, with three faces each and twelve *edge cubies*, with two faces each. The operators for this puzzle are defined to be, for each face of the cube, any 90 degree twist of the face clockwise or counterclockwise and a 180 degree twist. The state space for Rubik's Cube contains roughly $4 \times 10^{19}$ reachable states, and the median solution length, with this set operators, is believed to be 18 [17]. Two different state encodings have been used for Rubik's Cube in the past. We report experiments with both.

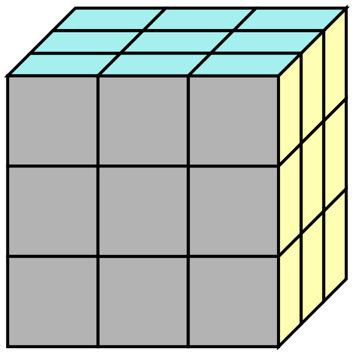


Fig. 3. Rubik's Cube

## *3.2 First Encoding Tested on "Easy" Instances*

In this section, the state encoding has separate sets of constants and separate variables for representing a cubie's identity and for representing its orientation. [5] This allows the orientation of a cubie to be abstracted while keeping its identity distinct.

All the abstractions in this experiment mapped all eight corner cubies to the same abstract constant $c$, effectively eliminating the corner cubies from the heuristic estimation of distance, they differ only in how they abstract the edge cubies. [6]

The large pattern database in this experiment is of size $m = 106,444,800$. The abstraction defining this pattern database, $PDB_1$, mapped four arbitrarily chosen edge cubies to the abstract constant $a$, and mapped three other arbitrarily chosen edge cubies to the abstract constant $b$. The remaining five edge cubies were kept unchanged.

We then experimented with maximizing over $n$ pattern databases of size $m/n$ for even values of $n$ ranging from 2 to 8. The abstractions for the two pattern databases for $n = 2$ mapped the same cubies to $a$ and to $b$ as in $PDB_1$ and, in addition, each abstracted away the orientation of one of the remaining edge cubies. The abstractions for the four pattern databases for $n = 4$ were defined in exactly the same way, except that each abstraction abstracted away the orientations of two edge cubies instead of just one. The abstractions for $n = 6$ each mapped three arbitrarily chosen edge cubies to the abstract constant $a$, three others to the abstract constant $b$, and three others to the constant $d$. The abstractions for $n = 8$ each mapped four edge cubies to the abstract constant $a$, and mapped four others to the abstract constant $b$.

| PDB Size | $n$ | Nodes Generated | Ratio (Nodes) | Seconds | Ratio (Seconds) |
|---|---|---|---|---|---|
| 106,444,800 | 1 | 61,465,541 | 1.00 | 141.64 | 1.00 |
| 53,222,400 | 2 | 5,329,829 | 11.53 | 14.35 | 9.87 |
| 26,611,200 | 4 | 3,096,919 | 19.85 | 10.55 | 13.43 |
| 17,740,800 | 6 | 2,639,969 | 23.28 | 9.90 | 14.31 |
| 13,305,600 | 8 | 2,654,689 | 23.15 | 11.72 | 12.09 |

Table 3
Rubik's Cube - results of the first experiment. "Easy" problem instances (solution length 12).

The various heuristics were evaluated by running IDA* on 10 problem instances

[5] An almost-identical encoding is described in detail in Section 4.4 of [11].
[6] While abstracting all corners may not yield the most informed heuristic function, our experimental comparisons remain meaningful since all abstractions shared this property.

whose solution length is known to be exactly 12. Table 3 shows the results of this experiment. The "Ratio" columns give the ratio of the number of nodes generated (or CPU time) using one large pattern database to the number of nodes generated (or CPU time) using $n$ pattern databases of size $m/n$. A ratio greater than one indicates that one large pattern database is outperformed by $n$ smaller ones.

The table shows that the number of nodes generated is smallest when $n = 6$, and is more than 23 times smaller than the number of nodes generated using one large pattern database. Similar experiments with thousands of states whose solution lengths ranged from 8 to 11 also exhibited reductions in nodes generated by a factor of 20 or more. These results show that maximizing over $n > 1$ pattern databases of size $m/n$ significantly reduces the number of nodes generated compared to using a single pattern database of size $m$. The same phenomenon has been observed consistently in smaller domains as well [14]. CPU time is also reduced substantially, although not as much as the number of nodes generated because consulting a larger number of pattern databases incurs a larger overhead per node.

## *3.3 Second Encoding Tested on "Easy" Instances*

In this section, the state encoding is the same as in [17], with a single state variable for each cubie, encoding both its position and orientation. Similar to the first experiment, the pattern databases in this experiment ignore the corner cubies altogether and are compared on "easy" problem instances. In particular, we compare a heuristic defined by a single pattern database based on 7 edge cubies to a heuristic defined by taking the maximum over 12 pattern databases based on 6 edge cubies each. Similar to the pattern databases built in [17] each of the 7 (or 6) edge cubies was mapped to a unique abstract constant while the rest cubies were mapped to a don't care constant $x$. Both heuristics require 255MB of memory in total. The heuristics were evaluated by running IDA* on 1000 problem instances, each obtained by making 14 random moves from the goal configuration (their average solution length is 10.66).

Table 4 presents the number of nodes generated and CPU time taken averaged over the 1000 test instances. Here we see again that using several small pattern databases outperforms one large pattern database: the number of nodes generated is reduced by a factor of 25 and CPU time is reduced by a factor of 10.

| PDB | Nodes | Time |
|---:|---:|---:|
| 1 7-edges | 90,930,662 | 19.6 |
| 12 6-edges | 3,543,331 | 1.9 |

Table 4
Average results on the 1000 easy Rubik's cube instances

| Problem | 8-corners + 7-edges | | 8-corners + 10 x 6-edges | | Ratio | Ratio |
|---|---|---|---|---|---|---|
| | Nodes | Time | Nodes | Time | (Nodes) | (Seconds) |
| 1 | 2,429,682,430 | 1,001 | 1,963,186,855 | 1,180 | 1.24 | 0.85 |
| 2 | 7,567,751,202 | 3,106 | 5,760,163,887 | 3,512 | 1.31 | 0.88 |
| 3 | 43,654,279,480 | 17,838 | 32,158,568,494 | 19,817 | 1.36 | 0.90 |
| 4 | 73,430,867,260 | 30,045 | 62,069,600,698 | 37,792 | 1.18 | 0.80 |
| 5 | 173,968,664,319 | 71,239 | 134,315,351,151 | 81,782 | 1.29 | 0.87 |
| 6 | 225,013,506,957 | 92,298 | 168,656,393,535 | 103,946 | 1.33 | 0.89 |
| 7 | 353,526,995,416 | 145,087 | 245,501,414,566 | 151,106 | 1.44 | 0.96 |
| 8 | 388,781,336,359 | 159,142 | 279,310,095,079 | 171,810 | 1.39 | 0.93 |
| 9 | 407,020,154,409 | 167,477 | 321,622,523,772 | 196,125 | 1.27 | 0.85 |
| 10 | 638,981,633,107 | 262,044 | 511,466,031,382 | 314,127 | 1.25 | 0.83 |

Table 5
Random instances for Rubik's cube

## *3.4 Second Encoding Tested on Random Instances*

In this experiment we use the same set of 10 random test instances as in [17], which have an average solution length of 16.4. We use the same 7-edge and 6-edge pattern databases as in the preceding experiment, but with each combined, by taking the maximum, with the pattern database based on 8 corner cubies used in [17], which requires an additional 255MB of storage.

The results are presented in Table 5 for each of these 10 instances. In terms of nodes generated, we again see that taking the maximum over several smaller pattern databases results in fewer nodes generated than using a single large pattern database. Here, however, the reduction is modest (between 18% and 44%) due to the fact that both heuristics also used the 8-corner pattern database, which had the maximum value in many cases. This modest reduction in nodes generated is not sufficient to compensate for the additional overhead required to access multiple pattern databases instead of just one, and therefore the better CPU time is obtained by using just one large pattern database for the edge cubies.

## 4 Maximizing after Adding

### *4.1 Additive Pattern Databases*

It is sometimes possible to add the heuristics defined by several pattern databases and still maintain admissibility. When this is possible, we say that the pattern databases are "additive."

For the sliding tile puzzles, pattern databases are additive if two conditions hold:

- The domain abstractions defining the pattern databases are disjoint [6,19]. A set of domain abstractions for the sliding tile puzzles is disjoint if each tile is mapped to $x$ ("don't care") by all but one of the domain abstractions. In other words, a disjoint set of domain abstractions for the sliding tile puzzles is defined by partitioning the tiles. A partition with $b$ groups of tiles, $B_1, ..., B_b$, defines a set of $b$ domain abstractions, $A_1, ..., A_b$, where domain abstraction $A_i$ leaves the tiles in group $B_i$ unchanged and maps all other tiles to $x$.
- The moves of a tile are only counted in the one pattern space where the tile is not mapped to $x$ ("don't care"). Note that this condition is not satisfied in the way pattern databases have been defined in previous sections. Previously, a pattern database contained the true distance from any given pattern to the goal pattern. All tile movements were counted, no matter what the tile was. Now only the movements of certain tiles are counted. This means the entries in additive pattern databases will often be smaller than the entries in normal pattern databases based on the same domain abstractions. It is hoped that the disadvantage of having slightly smaller entries is overcome by the ability to add them instead of merely taking their maximum.

If both the conditions are met, values from different pattern databases can be added to obtain an admissible heuristic. See [6,19] for more details on this method.

Given several different disjoint partitions of the tiles, the sum can be computed over the disjoint pattern databases defined by each partition, and then the maximum over these sums can be taken. We refer to this as "maximization after adding." The experiments in this section explore the usefulness of maximization after adding for the 15-puzzle and the 24-puzzle. As in the previous section, the experiments in this section compare the performance of IDA* obtained using several sets of smaller disjoint pattern databases with the performance obtained using one set of larger disjoint pattern databases.

| Heuristic | Memory | Nodes Generated | Ratio (Nodes) | Seconds | Ratio (secs) |
|---|---|---|---|---|---|
| one 7-7-1 | 115,315 | 464,977 | 0.29 | 0.23 | 0.48 |
| max-of-five(7-7-1) | 576,575 | 57,159 | 2.38 | 0.10 | 1.10 |
| one 8-7 | 576,575 | 136,288 | 1.00 | 0.11 | 1.00 |

Table 6
15-puzzle results.

### *4.2 Fifteen Puzzle*

Define an $x - y - z$ partitioning of the tile puzzle as a partition of the tiles into disjoint sets with cardinalities of $x$, $y$ and $z$. Consider an 8-7 partitioning for the 15-puzzle. The heuristic obtained is the sum of two pattern databases defined by partitioning the tiles into two groups, one with 8 tiles and the other with the other 7 tiles. The blank is considered a “don’t care” in both the 7-tile and 8-tile databases. Roughly speaking, the 8-tile database has 8 real tiles and 8 blanks, because moves of the “don’t care” tiles are not counted in this setting. [7] This is the best existing heuristic for this problem and was first presented in [19]. [8] We compare this with a heuristic we call the max-of-five(7-7-1) heuristic. This heuristic is defined by maximizing over 5 sets of disjoint pattern databases, where each set partitions the tiles into 3 groups, two of which contain 7 tiles while the third contains just one tile. Figure 4 shows the partition defining the 8-7 heuristic and one of the five 7-7-1 partitions used to define the max-of-five(7-7-1) heuristic.

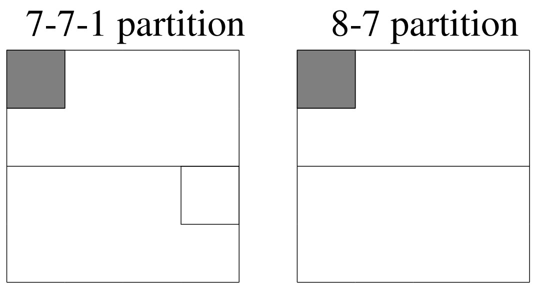


Fig. 4. Different partitions of the 15-puzzle tiles. The grey square represents the blank.

We ran these two heuristics on the 1000 random initial states used in [6,19]. Results for the different heuristics are provided in Table 6. Each row corresponds to a different heuristic. The first row gives the results for using only the 7-7-1 heuristic

[7] The exact way that the blank is treated is beyond the scope of this paper. A better discussion of the role of the blank in this setting can be found in [6].
[8] In [19], symmetries of the space are exploited so that the same pattern database can be used when the puzzle is reflected about the main diagonal. They took the maximum of the corresponding two heuristics without the need to store them both in memory. The experiments in this paper do not take advantage of these domain-dependent symmetries.

from Figure 4. The second row is for the max-of-five(7-7-1) heuristic. Finally, the bottom row gives the results for the 8-7 heuristic from Figure 4. The *Memory* column shows the memory requirements for each heuristic, which is the total number of entries in all the pattern databases used to define the heuristic. The *Nodes Generated* column gives the average number of nodes IDA* generated in solving the 1000 problem instances. The *Seconds* column provides the time in seconds it took to solve a problem, on average. The *Ratio* columns give the ratio of the number of nodes generated (or CPU time) by the 8-7 heuristic to the number of nodes generated (or CPU time) by the heuristic in a given row. A ratio greater than one means that the 8-7 heuristic is outperformed.

The results confirm the findings of the previous experiments. Maximizing over five different 7-7-1 partitionings generates 2.38 times fewer nodes than the 8-7 heuristic [9] and over 8 times fewer nodes than just one 7-7-1 heuristic. Note that the memory requirements for the 8-7 heuristic and the max-of-five(7-7-1) heuristic are identical. As was seen with Rubik's Cube, the CPU time improvement is smaller than the improvement in nodes generated.

The CPU time for solving the 15-puzzle with multiple sets of additive pattern databases can be reduced by the following observation. Since every operator only influences the tiles in one particular pattern database, for a given set of partitionings (e.g., five 7-7-1 partitionings), only one pattern database needs to be checked when moving from a parent node to a child node. The values of the other pattern databases within the same partitioning cannot change and need not be checked. Implementing this technique might need additional parameters to be passed from a parent node to a child node in IDA*. These parameters are the pattern database values of the unchecked pattern databases. The time and memory they require are negligible. This technique reduced the CPU time for the max-of-five(7-7-1) heuristic to 0.06 seconds and the time for the 8-7 heuristic to 0.07 seconds.

## *4.3 Twenty-Four Puzzle*

We have performed the same set of experiments on the larger version of this puzzle namely the 5x5, 24-puzzle. Here we tested two heuristics. The first one is called the 6-6-6-6 heuristic. This heuristic partitions the tiles into four groups of 6 tiles each. We used the same partitioning into groups of sixes that was used in [19]. We then partitioned the 24 tiles into 4 groups of five tiles and one group of four tiles. We call this the 5-5-5-5-4 heuristic. We have generated eight different 5-5-5-5-4 heuristics and combined them by taking their maximum. We call this heuristic the max-of-eight(5-5-5-5-4) heuristic. Note that the 6-6-6-6 heuristic needs $4 \times 25^6 = 976,562$

[9] Note that taking the maximum of two 8-7 partitioning by using the 8-7 partitioning of Figure 4 and its reflection about the main diagonal generated only 36,710 nodes in [19]. This provides additional evidence in favor of maximizing over several pattern databases.

Kbytes while the max-of-eight(5-5-5-5-4) heuristic needs $8 \times (4 \times 25^5 + 25^4) = 315,625$ Kbytes which is roughly a third of the 6-6-6-6 heuristic. [10]

Table 7 shows the number of generated nodes and CPU time required by IDA* using each of these heuristics to solve the first six problem instances from [19] with IDA*. The *Ratio* columns give the ratio of the number of nodes generated (or CPU time) for the 6-6-6-6 heuristic to the number of nodes generated (or CPU time) for the max-of-eight(5-5-5-5-4) heuristic. A ratio greater than one means that the max-of-eight(5-5-5-5-4) heuristic outperforms the 6-6-6-6 heuristic.

| Instance | one 6-6-6-6 | | max-of-eight(5-5-5-5-4) | | Ratio | Ratio |
|---|---|---|---|---|---|---|
| | Nodes | Seconds | Nodes | Seconds | (Nodes) | (Seconds) |
| 1 | 9,728,172,650 | 1,198 | 5,991,782,489 | 2,495 | 1.62 | 0.48 |
| 2 | 663,157,799,297 | 81,666 | 276,161,457,963 | 110,359 | 2.40 | 0.74 |
| 3 | 247,605,992,067 | 30,495 | 97,940,394,079 | 37,645 | 2.53 | 0.81 |
| 4 | 54,556,763,234 | 6,718 | 33,157,258,292 | 13,995 | 1.64 | 0.48 |
| 5 | 38,520,070,174 | 4,505 | 6,261,389,344 | 2,576 | 6.15 | 1.74 |
| 6 | 932,251,470,113 | 107,014 | 37,039,640,318 | 14,160 | 25.10 | 7.55 |

Table 7
24-puzzle results.

As can be seen, the number of nodes generated is always reduced by using the max-of-eight(5-5-5-5-4) heuristic. The amount of reduction varies among the different problem instances from 1.62 to 25.1. [11] In two instances CPU time is reduced by using multiple, smaller pattern databases, but in the other four it is increased.

## 5 Multiple Abstractions in Hierarchical Heuristic Search

Instead of building an entire pattern database before starting to solve any problem instances, it is possible to build the portion of the pattern databases needed to solve the instance on demand. This technique, called hierarchical heuristic search [13,15], is preferable to building the entire pattern database when there is only one or a small batch of problem instances to be solved.

[10] Using symmetric attributes of this domain such as reflection of the databases about the main diagonal as done in [19] could decrease these numbers for both heuristics. In this paper we decided not to exploit this domain-dependent property.

[11] For case 6 the number of generated nodes is better by a factor of 3 than the number of generated nodes reported by [19] (103,460,814,356), which used the 6-6-6-6 partitioning and its reflection about the main diagonal. For this particular instance we have the best published solution.

The results from the preceding experiments suggest that it might be advantageous, within the hierarchical heuristic search framework, to use multiple coarse-grained abstractions, corresponding to several smaller pattern databases, instead of using one fine-grained abstraction, corresponding to one large pattern database. However, the conclusion does not immediately carry over because in traditional pattern database studies the cost of building the pattern database(s) is not counted in the CPU time required to solve a problem, but in hierarchical heuristic search it must be. This means that it is possible, in hierarchical heuristic search, that the cost-per-node during search might be considerably higher using multiple abstractions than using a single abstraction. An experimental comparison of these two alternatives on four different state spaces was reported in [13]. The reader is referred to that paper for the definitions of the state spaces, the abstractions used, and other details.

Table 8 shows the total CPU time required by Hierarchical IDA* to solve a batch of 100 randomly generated test problems for each state space. As can be seen, using multiple coarse-grained abstractions has indeed proved useful for hierarchical heuristic search. Search with multiple coarse-grained abstractions is faster than using one fine-grained abstraction in all cases, almost twenty times faster for the 15-puzzle.

| State Space | One Abstraction | Multiple Abstractions | Ratio |
|---|---:|---:|---:|
| 15-puzzle | 59,600 | 3,100 | 19.2 |
| Macro-15 | 10,100 | 1,700 | 5.9 |
| (17,4)-TopSpin | 16,200 | 8,800 | 1.8 |
| 14-Pancake | 3,100 | 1,000 | 3.1 |

Table 8
Total time (in seconds) for Hierarchical IDA* to solve a batch of 100 problems.

## 6 Why the Number of Nodes Generated is Reduced by Maximization

When using just one pattern database, search performance in domains with uniform-cost operators is roughly proportional to the size of the pattern database, with larger pattern databases tending to outperform smaller ones [12]. The experiments in the previous sections have shown that this trend can be reversed by maximizing over several smaller pattern databases, provided they are not too small. Although individually inferior to a larger pattern database, they are collectively superior to it. Our explanation of this phenomenon is based on two conjectures, as follows.

**Conjecture 1:** Maximizing over several smaller pattern databases can make the number of states that are assigned low h-values significantly smaller than the number of states assigned low h-values using one larger pattern database.

This conjecture is intuitively clear. While small values exist in every pattern database, maximizing over the smaller pattern databases will replace small h-values by larger ones (unless all pattern database return small values for the specific state). This can substantially reduce the number of states that are assigned very small h-values. By contrast, if the large pattern database returns a small value, this small value is used.

**Conjecture 2:** Eliminating low h-values is more important for improving search performance than retaining large h-values.

This is not immediately obvious. To see why it is likely to be true, consider the formula developed in [20] to predict the number of nodes generated by one iteration of IDA* to depth $d$:

$$\sum_{i=0}^{d} N(i) \cdot P(d-i) \tag{1}$$

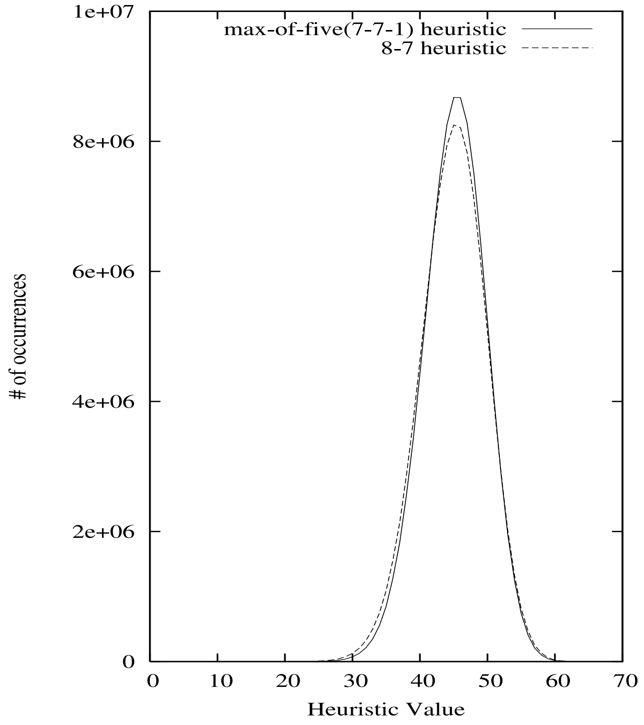
Here, $N(i)$ is the number of nodes at depth $i$ and $P(h)$ is the fraction of nodes with a heuristic value less than or equal to $h$. If two pattern databases differ only in that one has a maximum h-value of 11, while the other has a maximum value of 10, this has very little effect on the sum, since it only affects $P(11)$, $P(12)$, etc. and these are multiplied by $N$ values ($N(d-11)$, $N(d-12)$, etc.) that are typically relatively small. On the other hand, if two pattern databases differ in the fraction of nodes that have $h = 0$, this would have a large effect on the formula since this fraction is part of every $P(h)$, including $P(0)$ which is multiplied by $N(d)$, usually by far the largest $N$ value.

Conjecture 2 was first noted in [17]. Nodes with small h-values were observed to recur much more frequently in an IDA* search than nodes with high h-values. This is because if small h-values occur during search they do not get pruned as early as large values, and, if the heuristic is consistent, they give rise to more nodes with small h-values.

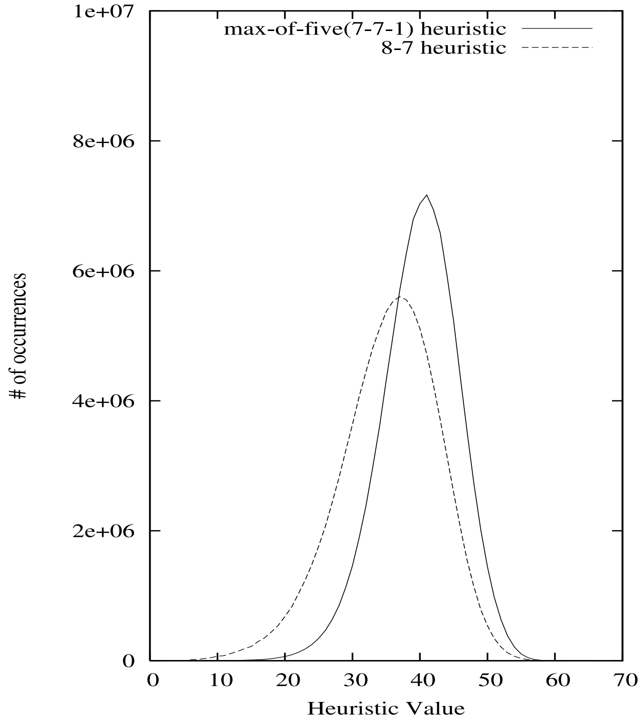
Together these two conjectures imply that the disadvantage of using smaller pattern databases – that the number of patterns with large h-values is reduced – is expected to be more than compensated for by the advantage gained by reducing the number of patterns with small h-values.

To verify these conjectures we created histograms showing the number of occurrences of each heuristic value for each of the heuristics used in our experiments. Figure 5 presents the histograms for the 15-puzzle heuristics.

Figure 5(a) shows the "overall" distribution of heuristic values for the 8-7 heuristic for the 15-puzzle (dashed line) and for the max-of-five(7-7-1) heuristic (solid line). These two histograms were created by generating 100 million random 15-puzzle



(a) Overall distribution



(b) Runtime distribution

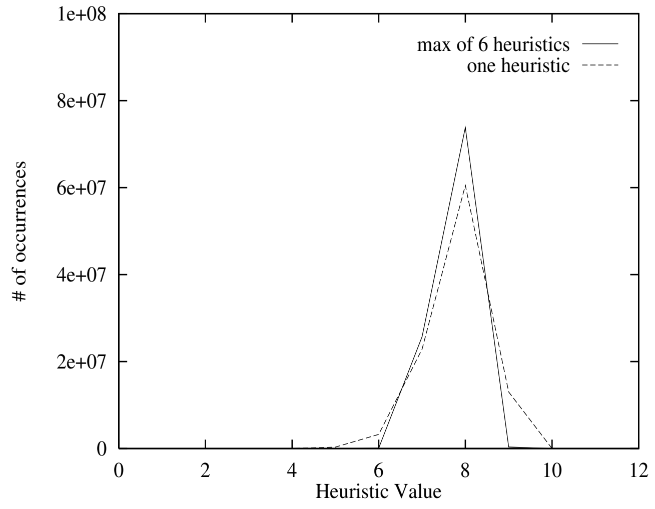
Fig. 5. Distributions of heuristic values for the 15-puzzle.

states and counting how many states had each different value for each heuristic. The two distributions are very similar, with averages of 44.75 for the 8-7 heuristic and 44.98 for the max-of-five(7-7-1) heuristic. As can be seen, the 8-7 heuristic has a greater number of high and low heuristic values (which confirms Conjecture 1 above) while the max-of-five(7-7-1) heuristic is more concentrated around its average.

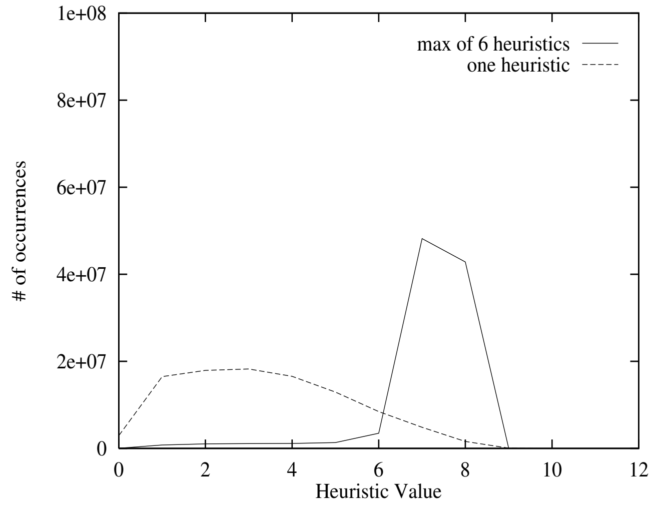
Figure 5(b) shows the "runtime" distribution of heuristic values for IDA* using each of these two heuristics. This histogram was created by recording the h-value of every node generated while running IDA* with the given heuristic on many start states and counting how many times each different heuristic value occurred in total during these searches. We kept on solving problems until the total number of heuristic values arising in these searches was 100 million, the same number used to generate the overall distributions in Figure 5(a). Unlike the overall distributions, there is a striking difference in the runtime distributions. Both distributions have shifted to the left and spread out, but these effects are much greater for the 8-7 heuristic than for the max-of-five(7-7-1) heuristic, resulting in the 8-7 heuristic having a significantly lower average and much greater percentage of low heuristic values. For example, IDA* search with the 8-7 heuristic generates twice as many states with a heuristic value of 36 or less than IDA* search with the max-of-five(7-7-1) heuristic. What seemed to be a relatively small difference in the number of low values in the overall distributions has been amplified during search to create a markedly different runtime distribution. This is empirical confirmation of the effects predicted by the above conjectures.

Figure 6 presents the analogous histograms for the Rubik's Cube experiment reported in Section 3.2. Exactly the same pattern can be seen. The overall distribution for a single large pattern database (the dashed line in Figure 6(a)) has a slightly higher average than the overall distribution for the max-of-6 smaller pattern databases (7.825 compared to 7.744) but is slightly less concentrated around the mean and has more low values. This causes its runtime distribution to be shifted much further to the left (Figure 6(b)).

Making the pattern databases too small has a negative impact on performance. This is because as the individual pattern databases become smaller the overall heuristic distribution shifts to the left, and eventually becomes shifted so far that the benefits of maximizing over multiple smaller pattern databases are outweighed by the losses due to the individual heuristics being extremely poor. For example, in the absolute extreme case of having $m$ pattern databases each of size 1, a heuristic value of zero would be returned for all states.



(a) Overall distribution



(b) Runtime distribution

Fig. 6. Distributions of heuristic values for Rubik's Cube.

## 7 Why Maximization can Fail

Table 9 shows the number of nodes generated by IDA* for each depth bound it uses when solving a particular 8-puzzle problem using three different heuristics, $h_1$, $h_2$, and $max(h_1, h_2)$. Heuristic $h_1$ is based on a domain abstraction that leaves the blank unique. These abstractions have the special property that distances in pattern space have the same odd/even parity as the corresponding distances in the original space. This has the consequence, well-known for Manhattan distance, that IDA*'s depth bound will increase by 2 from one iteration to the next. This can be seen in the $h_1$ column of Table 9 – only even depth bounds occur when using $h_1$.

Heuristic $h_2$ is different. The domain abstraction on which it is based maps a tile to the same abstract constant as the blank. Patterns in this pattern space have two blanks, which means distances in this pattern space are not necessarily the same

| depth bound | $h_1$ | $h_2$ | $max(h_1, h_2)$ |
|---|---|---|---|
| 8 | 19 | 17 | 10 |
| 9 | - | 36 | 16 |
| 10 | 59 | 78 | 43 |
| 11 | - | 110 | 53 |
| 12 | 142 | 188 | 96 |
| 13 | - | 269 | 124 |
| 14 | 440 | 530 | 314 |
| 15 | - | 801 | 400 |
| 16 | 1,045 | 1,348 | 816 |
| 17 | - | 1,994 | 949 |
| 18 | 2,679 | 3,622 | 2,056 |
| 19 | - | 5,480 | 2,435 |
| 20 | 1,197 | 1,839 | 820 |
| TOTAL | 5,581 | 16,312 | 8,132 |

Table 9
Nodes generated for each depth bound. ("-" indicates the depth bound did not occur using a given heuristic)

odd/even parity as the corresponding distances in the original space. Heuristic $h_2$ is still admissible (and consistent), but because it does not preserve distance parity, IDA*'s depth bound is not guaranteed to increase by 2 from one iteration to the next. In fact, it increases by one each time (see the $h_2$ column of Table 9).

If $h_1(s) \geq h_2(s)$ for all states $s$, $max(h_1, h_2)$ would behave identically to $h_1$. This is clearly not the case in this example, since on the first iteration (depth bound = 8) IDA* generates only 10 nodes using $max(h_1, h_2)$ compared to 19 when using $h_1$.

If there is a state, $s$, generated on the first iteration for which, $g(s) + h_1(s) = 8$ and $g(s) + h_2(s) = 9$, then the depth bound on the next iteration would be 9, as it is in Table 9. This is an entire iteration that is skipped when using $h_1$ alone. Thus, although $max(h_1, h_2)$ will never expand more nodes than $h_1$ or $h_2$ for a given depth bound, it might use a greater set of depth bounds than would happen if using just one of the heuristics on its own, and, consequently, might generate more nodes in total, as seen here.

A similar phenomenon was noted in Manzini's comparison of the perimeter search algorithm BIDA* with IDA* [21]. Manzini observed that BIDA* cannot expand more states than IDA* for a given depth bound but that BIDA* can expand more states than IDA* overall because “the two algorithms [may] execute different iterations using different thresholds” (p. 352).

## 8 Summary and Conclusions

In all our experiments we consistently observed that maximizing over $n$ pattern databases of size $m/n$, for a suitable choice of $n$, produces a significant reduction in the number of nodes generated compared to using a single pattern database of size $m$. We presented an explanation for this phenomenon, in terms of the distribution of heuristic values that occur during search, and provided experimental evidence in support of this explanation. We have also discussed the tradeoff between the reduction in the number of nodes generated and the increase in the overhead per node. Speedup in runtime can only be obtained when the node reduction is higher than the increase in the overhead per node. These results have immediate practical application – if node generation is expensive compared to pattern database lookups, search will be faster if several smaller pattern databases are used instead of one large pattern database. Finally, we showed that IDA*'s performance can actually be degraded by using a better heuristic, even when the heuristic is consistent.

There are many ways to continue this line of research. For example, there may be other ways to reduce the number of pattern databases consulted when multiple pattern databases are available. One technique that did not produce significant speedup in our experiments, but is worth further study, we call Consistency-based Bypassing. Because pattern databases define consistent heuristics, the values in the pattern databases for the children of state $s$ cannot differ by more than one from their values for $s$, and therefore pattern databases that had values for $s$ far below $h(s)$ need not be consulted in computing $h$ for the children of $s$.

Also, further research is needed to develop a better understanding of how to choose the best number and sizes of pattern databases. A first step towards this would be to investigate sets of pattern databases that are different sizes, for example, one or two larger pattern databases combined with several smaller pattern databases.

There are other approaches that were introduced in the past few years that improve search and pattern databases. For example, pattern databases can be compressed by only storing the minimum value among nearby entries. This compression significantly reduces memory requirements without losing much information [7]. A different method for reducing the memory needed for a pattern database is to store only the entries that might be needed to solve a specific given problem instance [25]. In some circumstances, it is possible to take the maximum over multiple lookups in the same pattern database [8]. Also there have been improvements to the basic search algorithms [1,18,23,24,26]. An important feature of these different approaches is that they are roughly orthogonal to one another, suggesting they can be used in combination to achieve a larger improvement than any of them individually. Future work can try to find the best way to combine them all or part of them.

## Acknowledgments

This research was supported in part by an operating grant and a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada and an iCore postgraduate fellowship. This research was also partly supported by NSF awards to Sven Koenig under contracts IIS-9984827 and IIS-0098807. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

## References

[1] A. Auer and H. Kaindl. A case study of revisiting best-first vs. depth-first search. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI-04)*, pages 141–145, August 2004.

[2] J. C. Culberson and J. Schaeffer. Efficiently searching the 15-puzzle. Technical report, Department of Computer Science, University of Alberta, 1994.

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

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

[5] S. Edelkamp and A. Lluch-Lafuente. Abstraction in directed model checking. In *Proceedings of the Workshop on Connecting Planning Theory with Practice, at the International Conference on Automated Planning and Scheduling*, pages 7–13, 2004.

[6] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, 22:279–318, 2004.

[7] A. Felner, R. Meshulam, R. C. Holte, and R. E. Korf. Compressing pattern databases. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 638–643, July 2004.

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

[9] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[10] I. T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. In *Proceedings of Operations Research 2003*, pages 355–362, 2003.

[11] I. T. Hernádvölgyi. *Automatically Generated Lower Bounds for Search*. PhD thesis, School of Information Technology and Engineering, University of Ottawa, 2004.

[12] I. T. Hernádvölgyi and R. C. Holte. Experiments with automatically created memory-based heuristics. *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA-2000), Lecture Notes in Artificial Intelligence*, 1864:281–290, 2000.

[13] R. C. Holte, J. Grajkowski, and B. Tanner. Hierarchical heuristic search revisited. *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA-2005), Lecture Notes in Artificial Intelligence*, 3607:121–133, 2005.

[14] R. C. Holte, J. Newton, A. Felner, R. Meshulam, and D. Furcy. Multiple pattern databases. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 122–131, June 2004.

[15] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 530–535, 1996.

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

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

[18] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 677–682, July 2004.

[19] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[20] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[21] G. Manzini. BIDA*: an improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.

[22] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-04)*, pages 497–511, 2004.

[23] U. Zahavi, A. Felner, R. Holte, and J. Schaeffer. Dual search in permutation state spaces. In *AAAI-06*, pages 1076–1081, 2006.

[24] R. Zhou and E. A. Hansen. Breadth-first heuristic search. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 92–100, June 2004.

[25] R. Zhou and E. A. Hansen. Space efficient memory based heuristics. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 677–682, July 2004.

[26] R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 683–688, July 2004.