Multiple Pattern Databases

R. C. Holte and **Jack Newton**
University of Alberta
Computing Science Department
Edmonton, Alberta, T6G 2E8, Canada
Email: {holte,newton}@cs.ualberta.ca

A. Felner and **R. Meshulam**
Computer Science Department
Bar-Ilan University
Ramat-Gan, Israel 92500
Email: {felner,meshulr1}@cs.biu.ac.il

David Furcy
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280, USA
Email: dfurcy@cc.gatech.edu

Abstract

A pattern database 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 domains 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.

Introduction

Algorithms A* (Hart, Nilsson, & Raphael 1968) and IDA* (Korf 1985) find optimal solutions to state space search problems. They visit states guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the actual distance from the initial state to the current state n and $h(n)$ is a heuristic function estimating the cost from n to a goal state. If $h(s)$ is "admissible" (i.e., it never overestimates the true distance to the goal) then these algorithms are guaranteed to find optimal paths to the goal from any state.

Pattern databases were introduced in (Culberson & Schaeffer 1994) as a method for defining heuristic functions and have proven very valuable. For example, they are the key breakthrough that enabled Rubik's Cube to be solved optimally (Korf 1997). Also, solutions to the sliding tile puzzles can be found much faster with pattern databases (Korf & Felner 2002). Pattern databases have also made it possible to very significantly shorten the length of solutions constructed using a macro-table (Hernádvölgyi 2001) and have proven useful in heuristic-guided planning (Edelkamp 2001).

A pattern database stores a heuristic function as a lookup table. To compute $h(s)$, state s is mapped to a pattern $\phi(s)$, which serves as an index into the table. The entry in the table for $\phi(s)$ is the heuristic value used for $h(s)$. The number of distinct patterns – the size of the pattern database – is generally much smaller than the number of states.

The method we use to define pattern databases (see the next section) is simple and has three important properties. First, it guarantees that the resulting heuristic functions are admissible.[1] 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. Finally, it makes it easy to define numerous different pattern databases for the same search space. The most successful applications of pattern databases have all used multiple pattern databases. For example, the heuristic function used to solve Rubik's Cube in (Korf 1997) is defined as the maximum of three pattern database heuristics. The best heuristic function for the 24-puzzle (Korf & Felner 2002) is defined using eight pattern databases. They are divided into two groups, with each group containing four pattern databases constructed in such a way that their values for any state can be added together without overestimating the true distance from the state to the goal. The heuristic function used is the maximum of the sum of the values in each group.

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 "max'ing" the two heuristics. h_{max} is guaranteed to be admissible (or consistent) if h_1 and h_2 are. An alternative way of combining heuristics is adding them. $h_{add}(s) = h_1(s) + h_2(s)$ is only admissible in special circumstances, but when it is admissible, it is considered "better" than h_{max} because $h_{add}(s) \geq h_{max}(s)$ for all s.

The ability to create multiple pattern databases of a wide variety of different sizes, and to combine them to create one heuristic, raises the following question, which is the focus of this paper: given m units of memory for storing pattern databases, how is this memory best used?

Our first set of experiments compares max'ing of 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 up to two orders of magnitude over $n = 1$ (one pattern database of size m).

Our second set of experiments investigates max'ing over additive groups of pattern databases. The performance of one additive group of maximum-size pattern databases is compared to the performance of two or more additive groups

[1] In fact, they have the stronger property of being consistent (Holte *et al.* 1996). This is especially important when using A*.

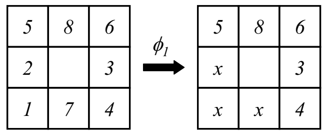

Figure 1: Abstracting a state makes a pattern

of smaller pattern databases. Here again the use of several, smaller pattern databases reduces the number of nodes generated.

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* expand more nodes using h_1 than it expands using h_2. The underlying cause of this behavior is explained.

Pattern Databases

The "domain" of a search space is the set of constants used in representing states. For example, the domain of the 8-puzzle might consist of constants $1 \dots 8$ representing the tiles and a constant, *blank*, representing the blank.

In (Culberson & Schaeffer 1998), 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 1 would be mapped to the pattern shown in the right part of Figure 1. The goal pattern is the pattern created by making the same substitution in the goal state.

The patterns are connected to one another using the same transition rules ("operators") that connect states. The "pattern space" formed in this way 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 corresponding patterns.

original	1	2	3	4	5	6	7	8	*blank*
ϕ_1	x	x	3	4	5	6	x	8	*blank*
ϕ_2	x	x	y	y	5	z	x	z	*blank*

Table 1: Examples of 8-puzzle domain abstractions.

A pattern database has one entry for every pattern in the pattern space. The value stored in the pattern database for pattern p is the distance in the pattern space from p to the goal pattern. Because the pattern space is an abstraction of the original state space, the pattern database defines an admissible heuristic function.

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 ϕ_1 of Table 1 shows the domain abstraction used above, 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 ϕ_2 in Table 1 shows this domain abstraction. Every different way of creating a row in Table 1 with 8 or fewer constants gives rise to a domain abstraction. This generalization is useful because it allows pattern databases of a greater variety of sizes to be created. As an illustration, consider the 15-puzzle, and suppose two gigabytes of memory are available for the pattern database. Using only the original types of "pattern," the largest pattern database that will fit in memory is based on an abstraction that maps 7 tiles to x ("don't care"). This uses only half a gigabyte (at one byte per entry) but the next larger pattern database, based on an abstraction that maps 6 tiles to x, requires too much memory (four gigabytes). However, an abstraction that maps 6 tiles to x and 2 tiles to y uses exactly two gigabytes of memory.

The "granularity" of a domain abstraction is a vector indicating how many constants in the original domain are mapped to each constant in the abstract domain. For example, the granularity of ϕ_2 is $\langle 3, 2, 2, 1, 1 \rangle$ because 3 constants are mapped to x, 2 are mapped to each of y and z, and constants 5 and *blank* each remain unique. In the experiments in this paper, the number of patterns produced by a domain abstraction is fully determined by its granularity.

Applying a domain abstraction to states is trivial, but constructing a pattern database also requires abstracting the operators. If the operators are expressed in an appropriate notation, such as PSVN (Hernádvölgyi & Holte 1999), they are as easy to abstract as states. An additional potential difficulty is that the *inverses* of the operators are needed in order to construct the pattern database in the most efficient manner, by running a breadth-first search backwards from the goal pattern until the whole pattern space is spanned. This issue, and one possible solution, is discussed in (Hernádvölgyi & Holte 1999).

Max'ing Multiple Pattern Databases

This section compares the performance of IDA* with heuristics defined using n pattern databases of size m/n for various values of n and fixed m. $h(s)$ is computed by max'ing the n pattern databases.

Experiments with Small Domains

The first set of experiments uses small search spaces so that tens of thousands of problem instances can be solved for each problem domain. $m = 5040$ was used for all experiments on these spaces. This size was chosen because it is a small percentage of the entire space and produces heuristics for the 8-puzzle that are almost as good as the Manhattan Distance heuristic. n varied from 1 (one pattern database of size 5040) to between 20 and 30, depending on the space. For each value of n, n abstractions of the same granularity were generated at random, producing a set, S, of n pattern databases whose sizes were all m/n. Each set S was evaluated by running IDA* with the heuristic defined by S on 100 problem instances whose solution lengths were all equal to the median solution length for the space. The same 100 instances were used in testing all the pattern database sets for a given domain.

Because of variability in average performance across different sets of the same size, we evaluated, for each n, 100 different sets of n pattern databases. This produced 100 averages for each n, with each average being over 100 problem instances. The number of nodes generated reported in the tables of this subsection is the overall average number of nodes generated by the 100 sets of pattern databases for each n.

Table 2 gives the results obtained for the 8-puzzle (median solution length of 22). It is clear that taking the maximum over two or more smaller pattern databases results in very significant reductions in nodes generated over using a single large pattern database. Using $n = 10$ reduces the number of nodes generated almost an order of magnitude over a single pattern database.

Granularity	PDB Size	n	Nodes Generated	CPU (secs)
$\langle 6,2,1\rangle$	252	20	585	0.04
$\langle 6,1,1,1\rangle$	504	10	460	0.02
$\langle 5,3,1\rangle$	504	10	725	0.03
$\langle 4,3,1,1\rangle$	2,520	2	1,212	0.02
$\langle 3,3,2,1\rangle$	5,040	1	3,842	0.07

Table 2: 8-puzzle results.

Granularity	PDB Size	n	Nodes Generated	CPU (secs)
$\langle 7,2,1,1,1\rangle$	47,520	21	2,203	0.22
$\langle 5,4,2,1\rangle$	83,160	12	3,511	0.23
$\langle 5,3,3,1\rangle$	110,880	9	3,600	0.20
$\langle 5,4,1,1,1\rangle$	166,320	6	4,659	0.19
$\langle 5,2,2,2,1\rangle$	498,960	2	20,658	0.55
$\langle 5,2,2,1,1,1\rangle$	997,920	1	408,666	11.72

Table 3: (3x4) sliding tile puzzle results.

A similar experiment was conducted using the (3x4) sliding tile puzzle. In this experiment $m = 997,920$, only 10 sets of pattern databases for each value of n were evaluated, and only 5 random problem instances having a solution length of 34 (the approximate median solution length for this space) were solved by each set of pattern databases. As before, randomly generated abstractions were used to define the pattern databases. As can be seen in Table 3, in this search space the reduction in nodes generated produced by max'ing over nine or more pattern databases is over two orders of magnitude.

Granularity	PDB Size	n	Nodes Generated	CPU (secs)
$\langle 6,2,1\rangle$	252	20	3,132	0.112
$\langle 6,1,1,1\rangle$	504	10	2,807	0.056
$\langle 5,3,1\rangle$	504	10	2,173	0.044
$\langle 4,3,1,1\rangle$	2,520	2	3,902	0.027
$\langle 3,3,2,1\rangle$	5,040	1	18,665	0.113

Table 4: 9-Pancake puzzle results.

Granularity	PDB Size	n	Nodes Generated	CPU (secs)
$\langle 5,2,1\rangle$	168	30	738	0.140
$\langle 3,3,2\rangle$	560	9	465	0.031
$\langle 2,2,2,2\rangle$	2,520	2	558	0.015
$\langle 2,2,2,1,1\rangle$	5,040	1	1,751	0.040
$\langle 5,2,1\rangle$	168	30	2,227	0.42
$\langle 3,3,2\rangle$	560	9	1,098	0.07
$\langle 2,2,2,2\rangle$	2,520	2	3,646	0.07
$\langle 2,2,2,1,1\rangle$	5,040	1	22,941	0.35

Table 5: (8,4)-TopSpin results. Upper 4 rows: 3-operator encoding. Lower 4 rows: 8-operator encoding.

To show that these results generalize to other spaces, analogous experiments were performed on the 9-pancake puzzle (median solution length = 8), and two encodings of the (8-4)-TopSpin puzzle, one with a low branching factor (3) and long solution lengths (median = 15) and one with a high branching factor (8) and shorter solution lengths (median = 8). These spaces are defined as follows.

In the N-pancake puzzle (Dweighter 1975), a state is a vector of length N containing N distinct values and there are $N-1$ operators, numbered 2 to N, with operator k reversing the order of the first k vector positions. We used $N = 9$, which has 9! = 362880 states.

The (N,K)-TopSpin puzzle has N tokens arranged in a ring. The tokens can be shifted cyclically clockwise or counterclockwise. The ring of tokens intersects a region K tokens in length which can be rotated to reverse the order of the tokens currently in the region. We used $N = 8$ and $K = 4$, which has 8! = 40320 states. We used two different encodings of this space. The first has 3 operators: (1) a circular shift 1 position clockwise, (2) a circular shift 1 position counterclockwise, and (3) a rotation of the special region. The second has N operators, one for each clockwise circular shift of length $0 \dots N-1$ followed by a rotation.

Tables 4 and 5 show the results, which confirm the previous findings. Max'ing over approximately ten small pattern databases outperforms the alternatives, reducing the number of nodes generated by an order of magnitude over a single pattern database in most of these domains.

Experiments with Rubik's Cube

Rubik's Cube is a standard benchmark domain for heuristic search. Random problem instances were first solved optimally by using the maximum of three pattern databases as a heuristic function (Korf 1997). Our experiments on Rubik's Cube provide additional support for the claim that max'ing n pattern databases of size m/n significantly reduces the number of node generations over using a single pattern database of size m.

Rubik's Cube has a much larger state space (there are 4.3252×10^{19} reachable states) than the previous domains. Therefore, the starting point for comparison in this experiment was one pattern database of size $m = 106,444,800$, the largest one that fit in our memory. The domain abstraction defining this pattern database was the following. Four of the edge cubies were mapped to the abstract constant a; three others were mapped to the abstract constant b; the remaining five edge cubies were kept unchanged; and all eight corner cubies were mapped to the same abstract constant c.[2] We then experimented with max'ing n pattern databases of size m/n for even values of n ranging from 2 to 8.

In addition to using more than one abstract constant to allow for fine-grained increments in the value of n, our state encoding kept the identity and orientation of the cubies separate. This allowed us to abstract the orientation of a cubie while keeping its identity distinct. We use a "1o" instead of a "1" to represent this case in the granularity vector.

Table 6 shows the number of nodes generated by IDA* for the chosen values of n (CPU times are reported in Table 11, below). Since the state space is so large, the corresponding heuristics were evaluated by running 10 problem instances whose solution length was 12 (the median solution length for Rubik's Cube is believed to be 18) with one set of pattern databases for each value of n. The table shows that node generations were reduced the most (by a factor of more than 23) with $n = 6$. Similar experiments (see the Appendix) with thousands of states whose solution lengths ranged from 8 to 11 also exhibited speedup factors of 20 or more. These results strongly support the claim that the number of node generations is reduced by using $n > 1$.

Granularity ⟨ corners ⟩⟨ edges ⟩	PDB Size	n	Nodes Generated
$\langle 8\rangle\langle 4,4,1,1,1,1\rangle$	13,305,600	8	2,654,689
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	17,740,800	6	2,639,969
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	26,611,200	4	3,096,919
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	53,222,400	2	5,329,829
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	106,444,800	1	61,465,541

Table 6: Rubik's Cube results.

Max'ing after Adding

As explained in the introduction, in some circumstances it is possible to add, instead of max'ing, the heuristics defined by several pattern databases and still maintain admissibility.

This happens, for example, with the sliding tile puzzles when the domain abstractions defining the pattern databases are disjoint (Korf & Felner 2002). 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. Another way to say this is that 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. With this formulation, if the moves of a specific tile are only counted in the one pattern space where the tile is not mapped to x, values from different pattern databases can be added to obtain an admissible heuristic. See (Korf & Felner 2002) for more details on this method.

Given several different 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 "max'ing after adding". The experiments in this section explore the usefulness of max'ing after adding for the 15-puzzle and the 24-puzzle. As in the previous section, the experiments in this section compare the performance (number of nodes generated by IDA*) obtained using several sets of smaller disjoint pattern databases with the performance obtained using one set of larger disjoint pattern databases.

Fifteen Puzzle

For the 15-puzzle, the heuristic defined by one set of larger disjoint pattern databases is called the 8-7 heuristic. It 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. This heuristic is the best existing heuristic for this problem and was first presented in (Korf & Felner 2002).[3] We compare this with a heuristic we call the max-of-five(7-7-1) heuristic. This heuristic is defined by max'ing 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 2 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.

We ran these two heuristics on the 1000 random initial states used in (Korf & Felner 2002). Results for the different heuristics on these 1000 instances are provided in Table 7 (CPU times are reported in Table 9, below). Each row corresponds to a different heuristic. For comparison reasons we give results for the Manhattan distance heuristic in the first row. The second row gives the results for the 7-7-1 heuristic from Figure 2. The third row is the max-of-five(7-7-1) heuristic. Finally, the bottom row gives the results for

[2]While abstracting all corners may not yield the most informed heuristic function, our experimental comparisons remain meaningful since all abstractions shared this property.

[3]In (Korf & Felner 2002), 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.

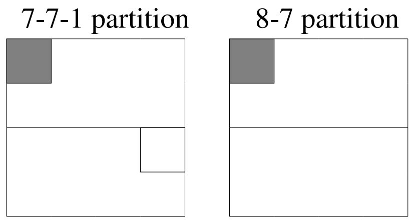

Figure 2: Different partitions of the 15-puzzle tiles. The grey square represents the blank.

Heuristic	Nodes Generated	Memory (Kbytes)
Manhattan	401,189,630	0
one 7-7-1	464,977	115,315
five 7-7-1	57,159	576,575
one 8-7	136,288	576,575

Table 7: 15-puzzle results.

the 8-7 heuristic from Figure 2. The *Nodes Generated* column gives the average number of nodes IDA* generated in solving the 1000 problem instances. 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 results confirm the findings of the previous experiments. One 7-7-1 heuristic generates 464,977 nodes, while max'ing 5 different 7-7-1 partitioning reduces that by a factor of almost 10. In contrast, the 8-7 heuristic generates more than twice as many nodes the max-of-five(7-7-1) heuristic[4]. Note that the memory requirements for the 8-7 heuristic and the max-of-five (7-7-1) heuristic are identical.

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 were used by (Korf & Felner 2002). 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 8 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 (abbreviated "eight (5-5-5-5-4)" in the Tables). Note that the 6-6-6-6 heuristic needs $4 \times 25^6 = 976,562$ 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.[5]

Run	6-6-6-6	eight(5-5-5-5-4)	Ratio
1	9,728,172,650	5,991,782,489	1.62
2	663,157,799,297	276,161,457,963	2.40
3	247,605,992,067	97,940,394,079	2.53
4	54,556,763,234	33,157,258,292	1.64
5	38,520,070,174	6,261,389,344	6.15
6	932,251,470,113	37,039,640,318	25.10

Table 8: 24-puzzle results.

Table 8 compares the number of generated nodes for these two heuristics when optimally solving the first six problem instances from (Korf & Felner 2002) with IDA* (CPU times are reported in Table 10, below). The *Ratio* column shows the number of nodes generated using the 6-6-6-6 heuristic divided by the number of nodes generated using the max-of-eight(5-5-5-5-4) heuristic. This is the speedup (in terms of nodes generated) that results from max'ing over eight groups of smaller additive pattern databases instead of using one group of large additive pattern databases. As can be seen, this speedup varies among the different instances from 1.62 to 25.1[6]. Thus, we can confirm that even for this domain, which is much larger than all the previous domains, the same benefits of max'ing occur.

Why Performance is Improved by Max'ing

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 (Hernádvölgyi & Holte 2000). The experiments in the previous sections have shown that this trend can be reversed if several smaller pattern databases are max'd together, providing 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 informal propositions, as follows.

Proposition 1: The use of smaller pattern databases instead of one large pattern database usually reduces the number of patterns with high h-values, and often reduces the maximum achievable h-value, but the max'ing of the smaller pattern databases can make the number of patterns with low h-values significantly smaller than the number of low-valued patterns in the larger pattern database.

The two assertions in this proposition are both intuitively clear. A smaller pattern database means a smaller pattern space, which typically means fewer patterns at all but the smallest distances, and therefore fewer patterns with high h-values. Max'ing the smaller pattern databases replaces small h-values by larger ones, and can substantially reduce the number of patterns with very small h-values.

[4]Note that taking the maximum of two 8-7 partitioning by using the 8-7 partitioning of Figure 2 and its reflection about the main diagonal generated only 36,710 nodes in (Korf & Felner 2002). This can be seen as another support to the advantage of max'ing.

[5]Using symmetric attributes of this domain such as reflection the databases about the main diagonal as done by (Korf & Felner 2002) could decrease these numbers for both heuristics. In this paper we decided not to exploit this domain-dependent property.

[6]For case 6 the number of generated nodes is better by a factor of 3 than the number of generated nodes reported by (Korf & Felner 2002) (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.

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

This assertion is not immediately obvious. To see why it is true, consider the formula developed in (Korf, Reid, & Edelkamp 2001) 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 level 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.

Together these two propositions 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.

Proposition 2 was first noted in (Korf 1997). 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 they give rise to more nodes with small h-values.

To verify these propositions we created histograms showing the number of occurrences of each heuristic value for each of the heuristics used in our experiments. Figures 3 and 4 present the histograms for the 15-puzzle heuristics. The histograms for the heuristics in other domains were similar.

Figure 3 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 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 slightly greater number of high and low heuristic values while the max-of-five(7-7-1) heuristic is more concentrated around its average.

Figure 4 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 3. 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 propositions.

Making the pattern databases too small has a negative impact on performance. This is because as the individual pattern databases become smaller the heuristic distribution

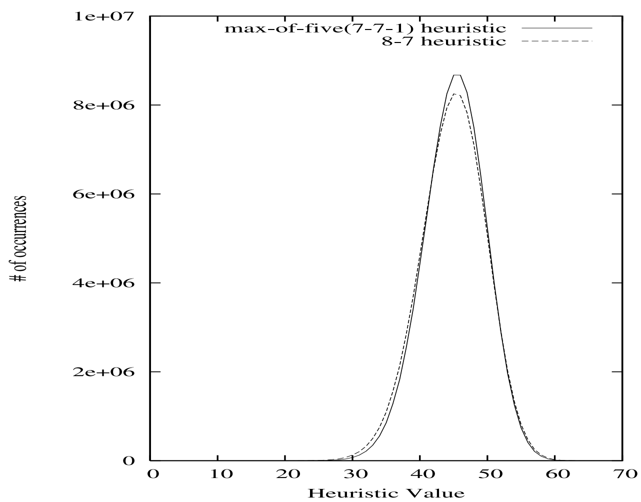

Figure 3: Overall distribution of heuristic values

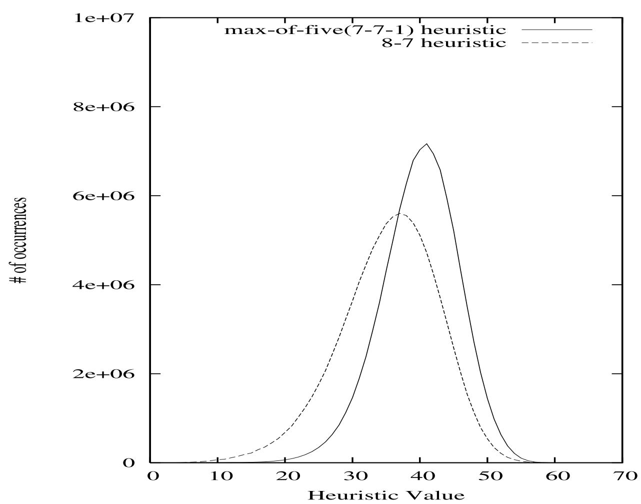

Figure 4: Runtime distribution of heuristic values

Heuristic	Seconds	Ratio (secs)	Ratio (Nodes)
one 7-7-1	0.23	0.48	0.29
max-of-five 7-7-1	0.10	1.10	2.38
one 8-7	0.11	1.00	1.00

Table 9: 15-puzzle timing results.

Run	6-6-6-6	eight(5-5-5-5-4)	Ratio (secs)	Ratio (Nodes)
1	1,198	2,495	0.48	1.62
2	81,666	110,359	0.74	2.40
3	3,049	3,764	0.81	2.53
4	6,718	13,995	0.48	1.64
5	4,505	2,576	1.74	6.15
6	107,014	14,160	7.55	25.10

Table 10: 24-puzzle timing results.

shifts to the left, and eventually becomes shifted so far that the benefits of max'ing are outweighed by the losses due to the individual heuristics being extremely poor.

The Overhead of Max'ing

Our experiments all show that using a heuristic defined by max'ing several pattern databases results in fewer nodes generated by IDA* than using a heuristic defined by one pattern database with the same total number of entries. However, this does not necessarily produce a corresponding reduction in runtime because computing a heuristic defined over several pattern database involves more pattern database lookups, and is therefore more time-consuming, than using just one pattern database. The overall speedup in runtime of IDA* thus depends on the relative magnitudes of the decrease in the number of nodes generated and of the increase in the cost of node generation.

Tables 9, 10, and 11 show the runtime in seconds of our experiments on the 15-puzzle, 24-puzzle and Rubik's Cube, respectively. "Ratio(secs)" is the number of seconds when using one large pattern database (or one set of large disjoint pattern databases) divided by the number of seconds when using several smaller pattern databases (or sets of smaller disjoint pattern databases). "Ratio(Nodes)" is the corresponding ratio of the number of nodes generated. These two ratios represent the average speedup (if greater than 1) or slowdown (if smaller than 1) as a multiplicative factor. The runtime speedup is always smaller than the node speedup because of the overhead of max'ing several pattern databases.

There is a simple enhancement that reduces the overhead of max'ing n pattern databases. The loop over the n pattern databases to compute the maximum can terminate as soon we encounter a heuristic value that makes the cost function $f = g+h$ exceed the current IDA* threshold. This node can be pruned immediately without the need to finish computing the maximum. This shortcut reduced the node overhead by a factor of more than 2 for Rubik's cube and by 60% for the tile puzzles. The times reported in this section were obtained using this shortcut.

Granularity	n	Seconds	Ratio (secs)	Ratio (Nodes)
$\langle 8\rangle\langle 4,4,1,1,1,1\rangle$	8	11.72	12.09	23.15
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	9.90	14.31	23.28
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	10.55	13.43	19.85
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	14.35	9.87	11.53
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	141.64	1.00	1.00

Table 11: Rubik's Cube timing results.

In the following analysis we will use n' to refer to the actual number of pattern databases consulted to compute the heuristic.

The overhead per node can be expressed as the sum of two factors, $C_h + C_o$, where C_h is the cost of computing the heuristic and C_o is the cost of all other IDA* operations such as applying the operators, determining the new threshold etc. C_h can be expressed as a product, $n' \times C_p$, where C_p is the cost of consulting one pattern database. The relation between node speedup and runtime speedup depends on n' and on algorithmic and implementation factors that influence the relative costs of C_p and C_o.

In the extreme case, the operations on the pattern databases (lookups, adding, and max'ing) dominate the other costs, $C_p \gg C_o$. In this case, the overhead per node when consulting n' pattern databases is almost n' times greater than the overhead per node of consulting one, and achieving a significant runtime speedup will therefore require a node speedup that is larger than n'.

For Rubik's Cube the bijective function used to map a state to an index in the pattern database is computationally expensive. Significant runtime speedups are possible because the node speedup is much larger than n'. In the tile puzzles, however, the node speedup (around 2.4) is too small to produce a significant runtime speedup for most of the cases. This is because the max-of-five (7-7-1) heuristic for the 15-puzzle involves roughly five times as many pattern database operations as the 8-7 heuristic and the max-of-eight(5-5-5-5-4) heuristic for the 24-puzzle involves roughly ten times as many pattern database operations as the 6-6-6-6 heuristic.

The overhead of the other basic operations of IDA* (C_o) is usually very small. Thus, any increase in time that is spent in calculating h (C_h) is directly reflected in the overall time per node. When using A*, by contrast, this will not be the case because of the substantial overhead in maintaining the Open list, which is part of C_o. Thus, any increase in C_h will not influence the overall time overhead as much. We therefore expect that max'ing might perform better when using A* than when using IDA*.

In the experiments on small domains, n pattern databases of size m/n were used. Given that each pattern database lookup incurs a cost, it is natural to ask if significant performance gains accrue if fewer than n pattern databases of size m/n are used. Although doing so will probably increase the number of nodes generated, that increase might be quite

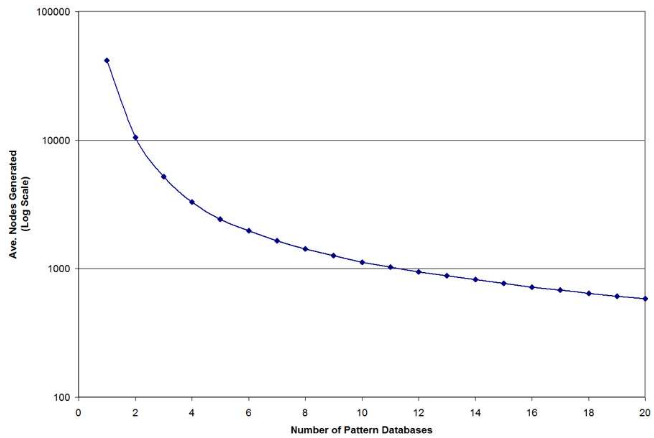

Figure 5: Nodes generated *vs.* number of pattern databases used (y-axis is a log scale).

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

small compared to the reduction in C_h, leading to an overall reduction in runtime. An initial exploration of this possibility is presented in Figure 5, which shows how the number of nodes generated decreases as more and more pattern databases are consulted. These results are for the 8-puzzle using $\langle 6, 2, 1 \rangle$ domain abstractions (each pattern database is size 252). Note that the y-axis (nodes generated) is on a log scale. As this Figure demonstrates, there are indeed diminishing returns for each additional pattern database used. One pattern database of size 252 performs very poorly. It is dramatically improved by adding a second pattern database of size 252. Adding the third and fourth pattern databases produces additional speedups, but the gains are clearly diminishing. Similar results were obtained for other domains. This suggests a promising direction for reducing the overhead incurred in max'ing multiple pattern databases.

Why Max'ing can Fail

Table 12 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 $max(h_1, h_2)$. h_1 and h_2 are both of granularity $\langle 3, 3, 2 \rangle$. h_1 is blank-preserving but h_2 is blank-increasing[7]. $max(h_1, h_2)$ will never expand more nodes than h_1 or h_2 for a given depth bound, but because it might use a greater set of depth bounds, and therefore do more iterations, its total number of nodes generated can be greater, as seen here.

This phenomenon was first noted in Manzini's comparison of the perimeter search algorithm BIDA* with IDA* (Manzini 1995). Manzini observed that BIDA* cannot expand more states than IDA* for a given *cost bound* but that BIDA* can expand more states than IDA* overall because "the two algorithms [may] execute different iterations using different thresholds" (p. 352).

[7] A domain abstraction is "blank-preserving" if it leaves the blank as a unique constant, distinct from the tiles, and is "blank-increasing" if it maps the blank and one or more tiles to the same abstract constant. Other than h_2 in this example, all abstractions for the sliding tile puzzles in this paper are blank-preserving.

Summary and Conclusions

In all our experiments we consistently observed that max'ing 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. Finally, we showed that IDA*'s performance can actually be degraded by using a better heuristic, even when the heuristic is consistent.

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.

Appendix: Additional Rubik's Cube Results

This appendix presents the results of two additional sets of experiments in the Rubik's Cube domain.

First, Tables 13 to 16 show the results of the same experiment as Table 6 for problem instances with solution lengths of 11, 10, 9, and 8, respectively. The same trends are apparent. Using several small pattern databases instead of one large pattern database reduces the number of nodes generated by a factor of 20 or more and reduces CPU time by a factor of about 10.

Second, Tables 17 to 19 show the performance of depth-bounded IDA* searches where the depth bound varied from 12 to 14, respectively. In these experiments, we used completely random instances whose solution length was not known a priori (as opposed to instances in the previous set of experiments) and turned out to be greater than 14. In other words, IDA* was executed on each instance until the threshold exceeded the given depth bound. Even though IDA* was stopped before finding a solution, these experiments allow us to measure the relative pruning power of our pattern database heuristics at different levels of the search tree. Again in all cases, we observe an order of magnitude speedup when using several small pattern databases instead of one large pattern database.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 4,4,1,1,1,1\rangle$	8	132,445	0.59
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	158,463	0.62
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	149,201	0.53
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	176,700	0.49
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	2,777,571	6.56

Table 13: Performance of full IDA* search averaged over 10 instances whose solution length equals 11.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 4,4,1,1,1,1\rangle$	8	16,258	0.07
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	18,126	0.07
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	18,273	0.06
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	25,327	0.07
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	366,394	0.84

Table 14: Performance of full IDA* search averaged over 100 instances whose solution length equals 10.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 4,4,1,1,1,1\rangle$	8	1,608	0.01
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	1,739	0.01
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	1,807	0.01
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	2,687	0.01
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	41,477	0.09

Table 15: Performance of full IDA* search averaged over 1000 instances whose solution length equals 9.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 4,4,1,1,1,1\rangle$	8	337	0.00
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	338	0.00
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	353	0.00
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	572	0.00
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	6,679	0.01

Table 16: Performance of full IDA* search averaged over 1000 instances whose solution length equals 8.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	2,560,295	10.36
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	2,406,539	8.70
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	2,507,068	7.25
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	36,502,603	85.15

Table 17: Performance of IDA* search to depth 12 averaged over 10 random instances.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	35,186,566	135.81
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	32,969,793	114.65
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	34,773,796	98.20
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	510,823,845	1,169.31

Table 18: Performance of IDA* search to depth 13 averaged over 10 random instances.

Granularity $\langle$ corners $\rangle\langle$ edges $\rangle$	n	Nodes Generated	CPU (secs)
$\langle 8\rangle\langle 3,3,3,1,1,1\rangle$	6	480,381,702	1,874.98
$\langle 8\rangle\langle 4,3,1o,1o,1,1,1\rangle$	4	449,985,200	1,617.39
$\langle 8\rangle\langle 4,3,1o,1,1,1,1\rangle$	2	477,980,524	1,355.48
$\langle 8\rangle\langle 4,3,1,1,1,1,1\rangle$	1	7,048,115,616	15,932.47

Table 19: Performance of IDA* search to depth 14 averaged over 10 random instances.

References

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

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

Dweighter, H. 1975. Problem e2569. *American Mathematical Monthly* 82:1010.

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

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

Hernádvölgyi, I. T., and Holte, R. C. 1999. PSVN: A vector representation for production systems. Technical Report TR-99-04, School of Information Technology and Engineering, University of Ottawa.

Hernádvölgyi, I. T., and Holte, R. C. 2000. 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.

Hernádvölgyi, I. T. 2001. Searching for macro operators with automatically generated heuristics. *Advances in Artificial Intelligence - Proceedings of the Fourteenth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (LNAI 2056)* 194–203.

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. *Proc. Thirteenth National Conference on Artificial Intelligence (AAAI-96)* 530–535.

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

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

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

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

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