Experiments with Automatically Created Memory-Based Heuristics

István T. Hernádvölgyi and Robert C. Holte

University of Ottawa
School of Information Technology & Engineering
Ottawa, Ontario, K1N 6N5, Canada
{istvan,holte}@site.uottawa.ca

Abstract. A memory-based heuristic is a function, $h(s)$, stored in the form of a lookup table: $h(s)$ is computed by mapping s to an index and then retrieving the corresponding entry in the table. In this paper we present a notation for describing state spaces, `PSVN`, and a method for automatically creating memory-based heuristics for a state space by abstracting its `PSVN` description. Two investigations of these automatically generated heuristics are presented. First, thousands of automatically generated heuristics are used to experimentally investigate the conjecture by Korf [4] that $m \cdot t$ is a constant, where m is the size of a heuristic's lookup table and t is the number of nodes expanded when the heuristic is used to guide search. Second, a similar large-scale experiment is used to verify that the Korf and Reid's complexity analysis [5] can be used to rapidly and reliably choose the best among a given set of heuristics.

1 Introduction

In this paper we describe a method for automatically creating heuristics from a description of a search space. The aim of this research is twofold. On the practical side, it is often difficult to generate good, provably admissible heuristics for a new search space. Our method is fully automatic and is guaranteed to generate monotone heuristics. On the scientific side, our method enables large-scale experiments to study properties of heuristics. For this purpose it is essential to create not just one heuristic for a search space, but many different ones whose properties can be controlled more or less directly by the experimenter. In this way general hypotheses about heuristics can be investigated experimentally.

Our general approach to automatically creating heuristics is to alter the description of the given search space, S, to create a description of a "simpler" search space, S', in such a way that (1) for every state in S, there is a corresponding state in S', and (2) the distance between any two states in S, is greater than or equal to the distance between the corresponding states in S'. A space with these two properties is called an abstraction of the original space [6]. Any abstraction of S gives rise to a monotone heuristic for searching in S: the distance between states s_1 and s_2 in S can be estimated by the exact distance between the corresponding states in S'.

For the purposes of automatically generating a wide variety of heuristics from a single search space description, and for having fine control over certain key features of the heuristics, we have found it useful to devise our own representation language, PSVN. To date we have studied one method of creating abstractions in PSVN, which we call domain abstraction. PSVN with domain abstraction generalizes the notion of *pattern database* [1]. Once the abstract space is created, the distance-to-goal for the entire abstract space is precomputed and stored in a lookup table with one entry for each abstract state. A heuristic represented by such a lookup table we call a memory-based heuristic.

The attraction of memory-based heuristics is that they enable search time to be reduced by using more memory. Korf [4] conjectures that memory (m) and time (t) can be directly traded off, *i.e.*, that the product $m \cdot t$ is a constant. This conjecture is important because if it is true search time can be halved simply by doubling available memory. In section 4.1 we test this conjecture in a large-scale experiment in which thousands of heuristics having a wide variety of memory requirements are evaluated. In section 5, a similar large-scale experiment is used to verify that the complexity analysis of search heuristics by Korf and Reid [5] can be used to rapidly and reliably choose the best among a given set of memory-based heuristics. Thus we can automatically generate a good heuristic for a novel search space by randomly generating a large set of heuristics and using Korf and Reid's method to select the best among them.

2 State Space Representation

To facilitate the automatic generation of many different abstractions of widely varying granularity, we use a simple vector notation for states and operators. A state is represented by a fixed length vector of labels from a finite set L called the domain. An operator is represented by a left-hand side (LHS) and right-hand side (RHS), each a vector the same length as the state vectors. Each position in the LHS and RHS vectors may be a constant (a label from L), a variable, or an underscore (_). The variables in an operator's RHS must also appear in its LHS. An operator is applicable to state s if its LHS can be unified with s. The act of unification binds each variable in LHS to the label in the corresponding position in s. Underscores in the LHS act as "don't cares". The RHS describes the state that results from applying the operator to s. The RHS constants and variables (now bound) specify particular labels and an underscore in a RHS position indicates that the resulting state has the same value as s in that position. For example,

$$< A, A, 1, \_, B, C > \rightarrow < 2, \_, \_, \_, C, B >$$

is an operator that can be applied to any state whose first two positions have the same value and whose third position contains 1. The effect of the operator is to set the first position to 2 and exchange the labels in the last two positions; all other positions are unchanged.

A state space is defined by a triple $S = < s_0, O, L >$, where s_0 is a state, called the *seed state*, O is a set of operators, and L is a finite set of labels. The state space is the transitive closure of s_0 and the operators, *i.e.*, it consists of all reachable states from s_0 by any sequence of operators.

We call this notation `PSVN` ("production system vector notation"). Although simple, it is expressive enough to specify succinctly all finite permutation groups (e.g. Rubik's Cube) and the common benchmark problems for heuristic search and planning (e.g. sliding tile puzzles).

3 State Space Abstraction

A *domain abstraction* is a map $\phi : L \to K$, where L and K are sets of labels and $|K| \leq |L|$. A *state space abstraction* is induced by a domain abstraction by applying ϕ to the seed state and the operators: $S' = \phi(S) = < \phi(s_0), \phi(O), K >$. The action of ϕ on an operator is to relabel the constants appearing in the operator. The abstract state space is defined to be the transitive closure of $\phi(O)$ and $\phi(s_0)$ – the set of states reachable from $\phi(s_0)$ by applying operators in $\phi(O)$. This definition extends the notion of "pattern" in the pattern database work [1], which in their framework is produced by mapping several of the labels in L to a special new label ("don't care") and mapping the rest of the labels to themselves.

The key property of state space abstractions is that they are homomorphisms and therefore the distance between two states in the original space, S, is always greater than or equal to the distance between the corresponding abstract states in $\phi(S)$. Thus, abstract distances are admissible heuristics for searching in S (in fact they are monotone heuristics: for formal proofs of these assertions see [2]).

The heuristic defined by an abstraction can either be computed on demand, as is done in Hierarchical A* [3], or, if the goal state is known in advance, the abstract distance to the goal can be precomputed for all abstract states and stored in a lookup table (pattern database) indexed by abstract states. In this paper we take the latter approach. If all the operators in S are invertible, the pattern database is constructed by an exhaustive breadth first traversal of S' starting at the goal state, $\phi(g)$, and using the inverses of the operators. If some operators are not invertible, the transpose of S' is created by a depth first forward traversal starting from $\phi(s_0)$ and then the pattern database is constructed by an exhaustive breadth first traversal of this explicit graph.

For special classes of search spaces a formula can be given relating an abstraction's *granularity* to the memory needed for the corresponding memory-based heuristic. But in general, the problem of estimating the size of the abstract space is difficult. The main complication is that an abstract space can contain an arbitrarily large number of states which have no pre-images. We call such an abstraction *non-surjective*. For example, consider the *2* × *2* sliding-tile puzzle and the domain abstraction $\phi_1 : \{0,1,2,3\} \rightarrow \{0,1,2\}$ defined as:

$$\phi_1(x) = \begin{cases} 0 \text{ if } x = 3 \\ x \text{ if } x \neq 3 \end{cases}$$

This abstraction has two 0's (blank tiles) as shown in Figure 1. It is non-surjective because there are states in $\phi_1(S)$ which have no pre-image in S. These states of $\phi_1(S)$ have dashed line boundaries in Figure 1.

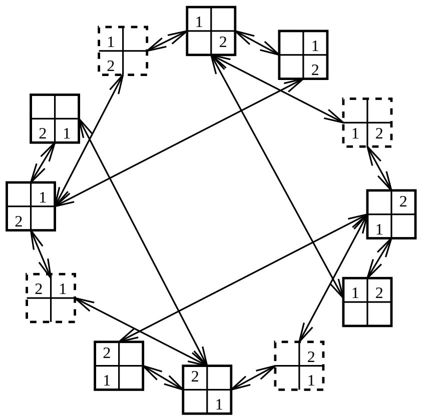

Fig. 1. $\phi_1(S)$

Non-surjective abstractions arise often in practice. All our attempts to represent the Blocks World in `PSVN` have given rise to non-surjective homomorphisms [2]. We have identified two causes of non-surjectivity, *orbits* and *blocks*. These are structural properties that naturally arise in problems in which the operators move physical objects (e.g. the cubies in Rubik's Cube) and there are constraints on which positions an object can reach or on how the objects can move relative to one another. We have also seen examples of other causes, but have not yet been able to give a general characterization of them.

4 Korf's Conjecture

A fundamental question about memory-based heuristics concerns the relationship between m, the size of the pattern database for a heuristic, and t, the number of nodes generated when the heuristic is used to guide search. [4] gives an insightful, but informal, analysis of this relationship which leads to the conjecture that $t \approx n/m$.

The aim of our first experiment is to examine the true relationship between t and m and compare it with the relationship conjectured in [4]. Our approach is to create abstractions with different values of m and problem instances with different values of d and measure t by running A* (not IDA*) with each abstraction on each problem instance. This is repeated for different search spaces to increase confidence in the generality of our conclusions. In these experiments all the abstractions are surjective, since Korf's conjecture is certainly false for non-surjective abstractions.

For a given m there can be many different abstractions. 30 are generated at random and their t values averaged. t is estimated separately for "hard", "typical", and "easy" problem instances using 100 randomly selected start states of each type (the goal state is fixed for each search space). The difficulty of a problem instance is determined by how its solution length compares to the solution lengths of all other problem instances. For example, we use the median of the solution lengths to define a "typical" problem instance.

	8-Puzzle	**8-Perm**	**Top-Spin**
n	181440	40320	40320
min m	252	56	28
max m	30240	20160	10080
b	1.667	6	2
d_{easy}	18	5	12
d_{typical}	22	7	16
d_{hard}	27	9	19

Table 1. Experiment Parameters

Results are presented (figure 2) as plots with m on the x-axis and t on the y-axis. Each data point represents the average of 3000 runs (30 abstractions, each applied to 100 problem instances). Breadth-first search was also run on all problem instances; it represents the extreme case when $m = 1$. In total, our experiments involved solving 236,100 problem instances. In this extended abstract we present only the results for the 8-puzzle.

We chose state spaces large enough to be interesting but small enough that such a large-scale experiment was feasible. Table 1 gives the general characteristics and experiment parameters for each space. Note that the m values for each space range from very small to a significant fraction of n. Each state space is generated by a puzzle, which we now briefly describe.

The 8-Puzzle is composed of 8 labeled sliding tiles arranged in a 3×3 grid. There is one tile missing, so a neighboring tile can be slid into its place. In PSVN each position in the vector corresponds to a particular grid position and the label in $vector[i]$ denotes the tile in the corresponding grid position. For example, if vector position 1 corresponds to the upper left grid position, and vector position 2 corresponds to the upper middle grid position, the operator that exchanges a tile in the upper left with an empty space (λ) in the upper middle is

$$< X, \lambda, \_, \_, \_, \_, \_, \_, \_ > \rightarrow < \lambda, X, \_, \_, \_, \_, \_, \_, \_ >$$

In the N-Perm puzzle a state is a vector of length N containing N distinct labels and there are $N-1$ operators, numbered 2 to N, with operator k reversing the order of the first k vector positions. We used $N = 8$. In PSVN operator 5, which reverses the first 5 positions, is represented

$$< A, B, C, D, E, \_, \_, \_ > \rightarrow < E, D, C, B, A, \_, \_, \_ >$$

The (N,K)-Top-Spin 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 $K = 4$, and three operators to define the state space

$$< I, J, K, L, M, N, O, P > \rightarrow < J, K, L, M, N, O, P, I >$$

$$< I, J, K, L, M, N, O, P > \rightarrow < P, I, J, K, L, M, N, O >$$

$$< A, B, C, D, \_, \_, \_, \_ > \rightarrow < D, C, B, A, \_, \_, \_, \_ >$$

4.1 Experimental Results

Figure 2 shows the experimental results for the 8-puzzle with m on the x-axis and t on the y-axis. The scale on both axes is logarithmic but the axes are labeled with the actual m and t values. With both scales logarithmic $t \cdot m =$ constant c, the conjecture in [4], would appear as a straight line with a slope of -1. Note that the y-axis is drawn at $m = 252$, not at $m = 0$.

In Figure 2 a short horizontal line across each line (at around $m = 4000$) indicates the performance of the Manhattan Distance on the 8-puzzle test problem instances. This shows that randomly generated abstractions of quite small size (5040 entries, less than 3% of the size of the state space) are as good as one of the best hand-crafted heuristics known for the 8-puzzle. The best of these randomly generated heuristics expands about 30% fewer nodes than the Manhattan distance.

A linear trend is very clear in all the results curves. The correlation between the data and the least squares regression line is 0.99 or higher in every case. However, the slope is not -1. These results therefore strongly suggest that $t \cdot m^{\alpha} =$ constant

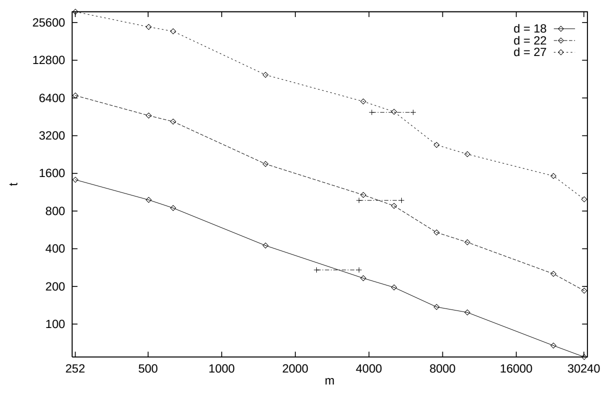

Fig. 2. 8-Puzzle: Number of States Expanded $[t]$ *vs* Size of Pattern Database $[m]$.

c for α between -0.57 and -0.8. α in this range means that doubling the amount of memory reduces the number of nodes expanded by less than a factor of 2.

Despite the very high correlation with a straight line, it appears that the top curves in each plot, and the middle curves to a lesser extent, are slightly bowed up, *i.e.*, that for problem instances with long solutions the effect on t of increasing m depends on the value of m, with a greater reduction in t being achieved when m is large. The reason for the flattening out of the curves as they approach $m = 1$ is a decrease in the effective branching factor as A* expands more states. A* caches all the states it generates. Search guided by a very small pattern database will generate a significant portion of the search space, and the more states generated the higher the chance that a freshly generated state is already in the cache. If plotted the effective branching factor of the 8-puzzle is seen to drop sharply as the size of the pattern database, m, decreases below 1000.

Figure 3 plots the average number of nodes expanded for every possible abstraction of the 8-puzzle in which the blank tile remains unique. The average is over 400 start states distance 22 from the goal state (a total of 1,392,400 problem instances were solved). There are some very small pattern databases – one of size 9 and eight of size 72. It is clear from that plot that for the search space for start states with distance 22 moves from the goal the linear trend continues along the entire scale of pattern databases. The plot also shows an interesting phenomenon: pattern databases of size 3024 slightly outperform pattern databases of size 3780, hence the notch in the curve.

Figure 4 shows how the pattern databases for the 8-puzzle of all different sizes perform on the same 400 start states. Clearly, the number of states expanded generally decreases as the size of the pattern database increases. However it is equally clear that there is significant overlap: the best heuristic with less memory is often better than an average heuristic with slightly more memory. On the other hand the range of variation for heuristics of the same size is not extremely large:

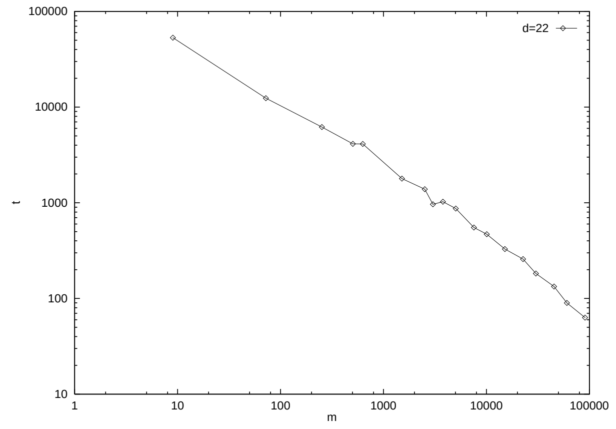

Fig. 3. 8-Puzzle – All Abstractions for $d = 22$: Number of States Expanded $[t]$ *vs* Size of Pattern Database $[m]$

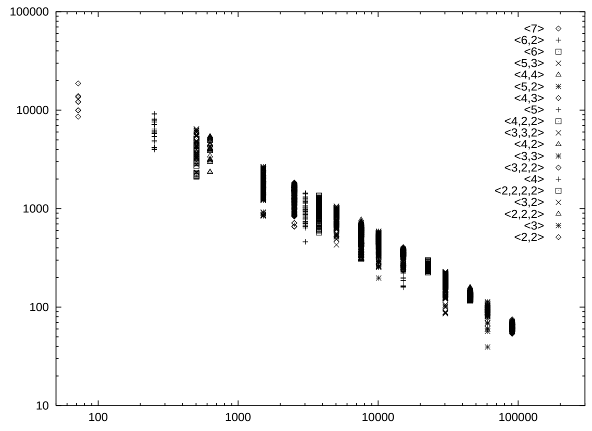

Fig. 4. Number of States Expanded by All Pattern Databases (y axis) *vs.* Memory (x axis). The Legend is Sorted in Increasing Memory Order. ($< a, b, ... >$ indicates that the domain abstraction ϕ was created by randomly choosing a many labels and assigning them the new label l_a, b many labels had new label image l_b ..., such that $l_a \neq l_b \neq$ .... The label representing the blank in all cases remained unique.)

the worst heuristic of any given size results in 2 to 4 times more nodes being expanded than the best heuristic of the same size.

5 Predicting a Heuristic's Performance

In [5] Korf and Reid develop a formula to predict the number of nodes generated, t, as a function of parameters that can be easily estimated for memory-based heuristics. Our reconstruction of their development, with some slight differences, leads to an estimate of the number of nodes

$$t(b,d) = \sum_{i=0}^{d} b^i \tilde{P}(d-i) \tag{1}$$

where b is the space's branching factor, d is the distance from the start state to the goal state, and $\tilde{P}(x)$ is the percentage of the entries in the pattern database that are less than or equal to x.

To verify how well equation 1 predicts the number of nodes expanded we have run an extensive experiment with the 8-Puzzle. The average number of nodes expanded for 400 start states (all distance 22 from the goal state) was measured for all abstractions that keep the blank tile unique. Each point in Figure 5 represents the average number of states expanded on these start states for one particular pattern database of size 5040. This actual number of states expanded is the x axis, the y axis is the value predicted by equation 1. If equation 1 precisely predicted the number of nodes expanded, the points would lie on the $y = x$ line.

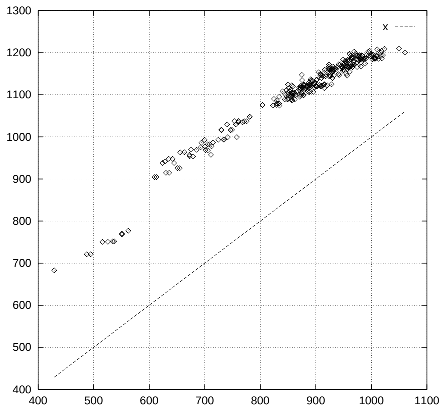

Fig. 5. Number of States Expanded as Predicted by Equation 1 (y axis) *vs.* The Average of the Actual Number of States Expanded (x axis). ($m = 5040$)

For all sizes of heuristic the relation between the actual and predicted values is an almost linear and certainly monotonic trend. Because of the monotonic relation between equation 1 and the actual number of nodes expanded, equation 1 can be used to reliably determine which of two heuristics will result in fewer nodes being expanded. The results shown here are when b and d are known exactly, but we also have additional experiments showing the same trends when b and d are estimated in a particular way.

6 Conclusion

In this paper we introduced a simple way of encoding problems, `PSVN`, in which states are vectors over a finite domain of labels and operators are production rules. By using `PSVN` domain abstractions are simple syntactic operations that result in monotonic search heuristics. Thus heuristics can be generated automatically from a state space's `PSVN` description. We report large scale experiments on moderate size search spaces to experimentally verify the relationship between a heuristic's memory and search effort and to predict the performance of pattern databases from the estimated branching factor of the search tree and the distribution of heuristic values in the pattern database.

6.1 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. Thanks to Jonathan Schaeffer and Joe Culberson for their encouragement and helpful comments and to Richard Korf.

References

1. J. C. Culberson and J. Schaeffer. Searching with pattern databases. *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)*, pages 402–416, 1996.
2. I. T. Hernádvölgyi and R. C. Holte. PSVN: A vector representation for production systems. Technical Report TR-99-04, School of Information Technology and Engineering, University of Ottawa, 1999.
3. 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.
4. 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.
5. R. E. Korf and Michael Reid. Complexity analysis of admissible heuristic search. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 305–310, 1998.
6. A. E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.