Steps Towards The Automatic Creation of Search Heuristics

István T. Hernádvölgyi[a] Robert C. Holte[b]

[a] *University of Ottawa, School of Information Technology & Engineering, Ottawa, Ontario, K1N 6N5, Canada,*

[b] *University of Alberta, Computing Science Department, Edmonton, Alberta, T6G 2E8, Canada,*

Abstract

The long-term goal of our research is to develop robust methods that use abstraction to create heuristics automatically from a description of a search space. Our research has progressed significantly towards this goal. This paper reviews the current state of the art, and the major open problems remaining to be solved.

Pattern databases are the foundation of the approach. The paper describes domain abstraction, which extends the notion of "pattern" in a way that permits available memory to be more fully exploited to reduce search time.

The paper demonstrates that a certain easily computed approximation to a formula developed by Korf and Reid [20] is monotonically related to the actual number of nodes expanded using the pattern database. This involves a large-scale experiment involving all possible domain abstractions for the 8-Puzzle in which the blank tile remains unique.

The principal remaining obstacle to automatic heuristic creation is shown to be the difficulty of predicting or controlling the size of a pattern database. This difficulty arises for two reasons, the main one being "non-surjectivity": domain abstractions can create abstract spaces in which some states do not have a pre-image. The paper identifies two specific causes of non-surjectivity, related to the space's orbits and blocks, and illustrates others.

[1] This research was supported in part by an operating grant and a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

The long-term goal of our research is to develop robust methods that use abstraction to create heuristics automatically from a description of a search space. There are both practical and scientific motives for this research. On the practical side, it is often difficult to hand-craft good, provably admissible heuristics for a new search space. This human effort, and the potential for human error, would be obviated by a fully automated method, and a method based on abstraction would be guaranteed to generate monotone heuristics, not just admissible ones. On the scientific side, such a method would enable 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 research has progressed very considerably towards this goal. This paper reviews the current state of the art, and the major open problems remaining to be solved.

Our general approach to creating heuristics automatically is the same as others who have studied this problem. The description of the given search space, S, is altered to create a description of a "simpler" search space, S'. Care is taken to ensure the new space has certain desirable properties. In particular, for every state in S, there must be a corresponding state in S' (but one state in S' might correspond to many different states in S), and the distance between any two states in S, must be greater than or equal to the distance between the corresponding states in S'. A space with these properties is called an abstraction of the original space [23]. 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'.

In particular, we follow the *pattern database* approach[4], which has been instrumental in defining sufficiently accurate and cost-effective heuristics to solve very large state spaces (Rubik's Cube [19], the 15-Puzzle [4]). In this approach, an abstraction is defined by treating some of the distinct values in the definition of the search space as if they were identical. Our version of this intuitive idea, which we call *domain abstraction*, generalizes the notion of "pattern" in the pattern database work. Once an abstraction has been chosen, the distance-to-goal for the entire abstract space is precomputed and stored in a lookup table – the pattern database – with one entry for each abstract state. To calculate $h(s)$, the heuristic value for state $s \in S$, one simply looks up the entry for the abstract state corresponding to s.

Prior work on pattern databases has not been concerned with the automatic creation of heuristics. The focus has been on hand-crafting heuristics to solve particular problems, and the pattern database framework was developed, and has proven highly successful, for quickly hand-crafting very effective heuristics.

However, by their very nature, pattern databases provide an ideal framework for creating heuristics automatically. The space of all possible patterns, or domain abstractions, is easy to represent and enumerate.

A simple local-search algorithm for searching through the space of pattern databases is given in Figure 1. More sophisticated search methods are possible, of course, but this suffices to provide a framework in which to understand the main topics discussed in the paper.

Given:

- m, the maximum size of the pattern database
- $score$, a function computing the quality of a pattern database
- S, a state space
- D, the domain for S
- $goal$, a goal state

Return:

- D', an abstract domain ($|D'| < |D|$)
- $\phi : D \rightarrow D'$, a domain abstraction
- $S' = \phi(S)$, the abstract state space induced by ϕ on S
- PDB, the pattern database for S' (a heuristic for S)

```
(1) initialize $\phi$
(2) Repeat
    (a) enumerate $S' = \phi(S)$ to create $PDB$
    (b) If the enumeration fails (because $|S'| > m$) make
        $\phi$ more abstract
    (c) If the enumeration succeeds ($|S'| \leq m$):
         (i) compute $score(PDB)$
        (ii) if $|S'| < m$: make $\phi$ less abstract
       (iii) if $|S'| = m$: change $\phi$ randomly
(3) return the highest-scoring $PDB$ with $|PDB| \leq m$
```

Fig. 1. Pseudocode for the automatic generation of heuristics. See the text for details.

Section 2 introduces our method of abstraction – domain abstraction – and pattern databases. Section 3 is concerned with the scoring function, $score$, used in Figure 1. Ideally the highest scoring pattern database would define the heuristic that resulted in the least search effort. A natural candidate is the formula derived by Korf and Reid [20] for the complexity analysis of search heuristics. This formula is introduced and evaluated as a scoring function in a large-scale experiment involving thousands of automatically created heuristics.

Section 4 introduces the problem of finding a domain abstraction whose corresponding pattern database is of a certain size. This is important because the pattern database must fit in the memory allocated for it. The non-trivial nature of this problem is reflected in Figure 1 by the fact that the size can only be determined by enumerating the pattern database (aborting if it exceeds the maximum allowable size). Section 4 also introduces heuristic methods for increasing or decreasing "abstractness" required in steps 2(b) and 2(c)(iii) of Figure 1 and discusses other aspects of this search procedure.

Section 5 continues the discussion of the issue of pattern database size, introducing a complication, *non-surjectivity*, that is the major remaining obstacle to fully automatic generation of pattern database heuristics.

Section 6 illustrates what is currently possible. First it uses two novel search spaces – the Pyraminx and the Skewb Cube puzzles – to illustrate how non-surjectivity can be avoided by human analysis and guidance of the heuristic-creation process. Second it describes a successful application of a simplified version of Figure 1 to the problem of finding optimal-length macro-operators for Rubik's Cube[11].

1.1 Previous Work

The earliest work on automatically creating heuristics ([8], [9], [22]) proposed using abstractions in which S and S' have precisely the same set of states but S' is more richly interconnected. If search spaces are described in a STRIPS-like language, this type of abstraction can be created by dropping preconditions from operators, or by adding macro-operators to the original set of operators. Gaschnig [8] calls such an abstraction an "edge supergraph"; Pearl [22] calls it a "relaxed problem". We will refer to such abstractions as embeddings, since the original search space is embedded in the abstract space.

This early work focused on hand-worked demonstrations that many existing heuristics could be created in this way. Automatic methods were not implemented. Nevertheless, there was an immediate concern about the efficiency of doing a separate, complete search every time a heuristic value was needed. Even if these searches were in a "simpler" space, it was feared that their total cost would outweigh the benefits of having a heuristic to guide the search in the original space. A heuristic whose cost to compute outweighs its benefits is called ineffective. Valtorta [24] proved that if the abstraction used to define a heuristic is an embedding the heuristic is always ineffective: every node expanded by blindly searching in the original space will also be expanded if the heuristic is computed by blindly searching in the abstract space. In short, embeddings on their own cannot be used to create effective heuristics. Gaschnig

[8] did not address this difficulty; in his only example he hand-crafted a very efficient algorithm for finding solutions in the abstract space. Guida and Somalvico [9] suggested caching information generated during every search in the abstract space and using it to speed up subsequent searches in the abstract space. Pearl [22] proposed searching for abstractions having special properties that can be automatically detected and exploited to ensure distances in the abstract space can be computed efficiently. Both the latter ideas proved successful in subsequent research, but in conjunction with alternative ways of doing abstraction ([23], [21], [14]).

A more promising type of abstraction is the homomorphism, which maps several different states in S to the same state in S' thereby enabling the abstract space to be considerably smaller than the original space. This type of abstraction was first identified by Kibler [16] and Banerji [1]. A version of Valtorta's theorem applies to all types of abstraction [14] but for homomorphisms the theorem gives clear hope for substantial speedup, and, indeed, Hierarchical A* [14] achieved speedup using homomorphic abstractions with the heuristic values computed on demand by searching in the abstract space.

ABSOLVER ([23], [21]) was the first system to automate all aspects of this general approach to creating heuristics automatically. ABSOLVER works with a restricted STRIPS representation and has seven different abstraction methods, each of which can be applied to a given search space description in a variety of ways. This, together with the fact that the composition of two abstractions is also an abstraction, creates a rich space of possible abstractions. ABSOLVER searches this space for abstractions to which it can apply one or more of its six speedup transformations. The latter are deemed necessary because it is felt that without them computing heuristic values on demand by searching in an abstract space will be too inefficient. ABSOLVER was tested on 13 different domains and automatically created heuristics for all of them. Slightly over half of these could be computed efficiently enough to speed up the overall search process.

One of ABSOLVER's speedup transformations is to precompute the distance-to-goal for the entire abstract space and store the result in a lookup table with one entry for each abstract state. With such a table the heuristic value for a state s is computed by determining the abstract state that corresponds to s and then looking up the heuristic value in the table. The Centre-Corner heuristic ABSOLVER discovered for Rubik's Cube and the X-Y heuristic for the 8-puzzle were made effective in this manner, leading ABSOLVER's creators to observe that "abstraction coupled with precomputation can produce effective heuristics".

In fact, this statement underestimates the power of this technique. Schaeffer and Culberson independently developed the "pattern database" technique

([3], [4]), and used it to define very effective heuristics for the 15-puzzle. Korf subsequently used pattern databases to define heuristics that enabled randomly generated problems for Rubik's Cube to be solved efficiently for the first time [19], and most recently Edelkamp has successfully applied pattern databases to classical planning problems[6]. A pattern database is precisely a precomputed distance-to-abstract-goal lookup table based on a special form of homomorphic abstraction.

The work presented in this paper extends the pattern database work in two ways. First, our notion of domain abstraction generalizes the type of abstraction used to define a pattern database. Second, we present a method for automatically creating abstractions whereas pattern databases have always previously been handcrafted.

2 Using Abstraction to Create Pattern Database Heuristics

2.1 State Space Representation

A *domain* is a finite set of values called *labels*. Formally, the labels of domain D are denoted by subscripting D (D_0, D_1 ... $D_{|D|-1}$), but when it is unambiguous to do so, we will use the integers on their own to denote the labels (e.g. we will write 1 instead of D_1).

A state "over domain D" is an instance of any data structure in which the entries labels of domain D.

A state space is defined by a domain, D, a state, *seed*, over D called the "seed state", and a successor function, *succ*, which, given any state s, returns the (possibly empty) set of states (over D) that can be reached directly from s. The state space is the transitive closure of *succ* starting with its application to *seed*. The notation $S =< D, seed, succ >$ indicates the domain, seed state, and successor function defining state space S.

States spaces will be defined in this paper using a simple vector notation which we call `PSVN` ("production system vector notation")[12]. In `PSVN` a state is a fixed-length vector whose values are all drawn from the given domain, and a successor function is a set of operators, where each operator is a production rule in which the left-hand side (LHS) and right-hand side (RHS) are each a vector the same length as the state vectors. Each position in the LHS and RHS vectors may be a constant (a label), 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.

This notation is used simply for presentation purposes; none of the results or discussion hinges upon its use.

In this paper we assume that the successor relation is symmetric, i.e., $s_1 \in succ(s_2)$ if and only if $s_2 \in succ(s_1)$. We also assume uniform (unit) costs of proceeding from any state to any of its successors.

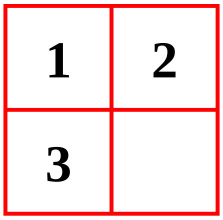


Fig. 2. *2* × *2* Sliding Tile Puzzle

The running example we will use for illustration throughout much of the paper is the *2* × *2* sliding-tile puzzle depicted in Figure 2. The unoccupied position, which we will regard as containing a blank tile, can exchange places with either neighbouring tile. The number of states reachable from the state in Figure 2 is 12. The state space is shown in Figure 3.

One natural `PSVN` representation of this space has a vector position for each of the four puzzle positions, with the value in vector position i indicating which tile is in the corresponding puzzle position. The domain in this case would have 4 labels, $\{0, 1, 2, 3\}$, with 0 representing the blank tile and labels 1-3 representing the corresponding non-blank tiles. If vector positions 1-4 represent puzzle positions “top left”, “top right”, “bottom left”, and “bottom right” respectively, the state in Figure 2 would then be represented as $< 1, 2, 3, 0 >$.

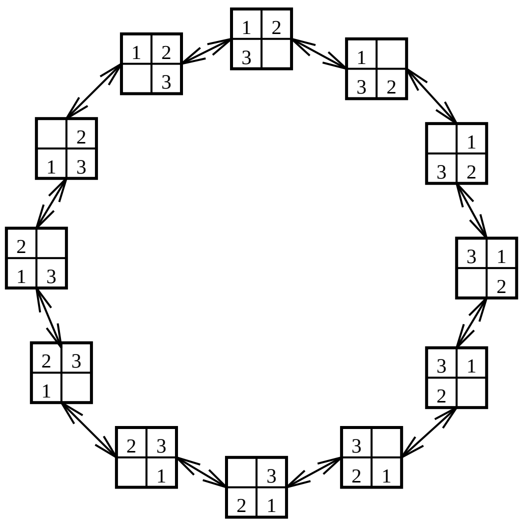


Fig. 3. The *2* × *2* Sliding Tile Puzzle State Space

2.2 State Space Abstraction

Definition 1 *Domain Abstraction*

A domain abstraction is any mapping $\phi : D \to D'$, where $|D'| < |D|$.

For example, one possible domain abstraction for the *2* × *2* sliding-tile puzzle is $\phi_1 : \{0, 1, 2, 3\} \to \{0, 1\}$ defined as:

$$\phi_1(0) = 0$$
$$\phi_1(1) = \phi_1(2) = \phi_1(3) = 1$$

In this mapping, the blank tile remains unique (label 0) but the non-blank tiles are all mapped to the same abstract label (1), so they are still distinguishable from the blank but are no longer distinguishable from each other.

The fact that three original labels map to one of the abstract labels and one original label maps to the other abstract label is an important characteristic of this abstraction. We say ϕ_1 has a "granularity" of $< 3, 1 >$. The general definition of granularity is as follows.

Definition 2 *Granularity*

The "granularity" of domain abstraction $\phi : D \to D'$ is defined to be a vector, $GRAN^{\phi}$, of length $|D'|$, with $GRAN^{\phi}_i = g_i$, the number of labels in D that

ϕ maps to label D'_i. Without loss of generality we assume that $GRAN^{\phi}_i \geq GRAN^{\phi}_{i+1}$ for all i.

For example, $GRAN^{\phi_1} =< 3, 1 >$ as just discussed. The domain abstraction

$$\phi_2(0) = \phi_2(3) = 0$$
$$\phi_2(1) = 1$$
$$\phi_2(2) = 2$$

maps two of the original labels to one abstract label (0) while the other two original labels remain unique. Thus $GRAN^{\phi_2} =< 2, 1, 1 >$. The domain abstraction

$$\phi_3(0) = \phi_3(1) = 0$$
$$\phi_3(2) = \phi_3(3) = 1$$

has $GRAN^{\phi_3} =< 2, 2 >$.

The 1's in $GRAN^{\phi}$ indicate labels that remain unique in the abstract space. For brevity, when $|D|$ is clear from the context we omit the 1's from the granularity vector (since the vector elements must add up to $|D|$, the number of 1's can be determined knowing the other values). Thus for the *2* × *2* puzzle we will normally write $GRAN^{\phi_1} =< 3 >$ instead of $< 3, 1 >$ and $GRAN^{\phi_2} =< 2 >$.

Definition 3 *State Space Abstraction Induced by a Domain Abstraction*

If state space $S =< D, seed, succ >$ and $\phi : D \to D'$ is a domain abstraction, the abstract state space induced by ϕ on S, denoted $\phi(S) =< \phi(D), \phi(seed), \phi(succ) >$, is the transitive closure of $\phi(succ)$ applied to $\phi(seed)$.

For this definition to be complete, it is necessary to explain what it means to apply ϕ to a state and to a successor function. If $\phi : D \to D'$ and s is any state over domain D, $\phi(S)$ is the state (over D') constructed by applying ϕ to each label in s.

The property required of the abstract successor function, $\phi(succ)$, is that it preserve the successor relation defined by $succ$ in the sense that if s_2 is a successor of s_1 then $\phi(s_2)$ is a successor of $\phi(s_1)$ [2]. If successor functions are written in PSVN, $\phi(succ)$ can be computed simply by applying ϕ to each label in $succ$[12].

[2] this statement of what is required assumes costs are uniform

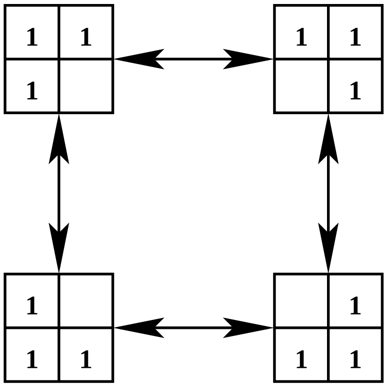


Fig. 4. Abstract space induced by ϕ_1

For example, the state space abstraction induced by the domain abstraction ϕ_1 (defined above) on S = the *2 × 2* puzzle is shown in Figure 4. $\phi_1(S)$ has four states, with three states of S mapped to each abstract state.

These definitions guarantee that the distance between any pair of states in S is greater than or equal to the distance between the corresponding states in $\phi(S)$. Thus, abstract distances are admissible heuristics for searching in S (in fact they are monotone heuristics [14]).

Definition 4 *Heuristic based on a Domain Abstraction*

Let S be a state space over domain D, g a state in S, and $\phi : D \to D'$ a domain abstraction. Then the heuristic, $h_{\phi,g}$, for goal g based on ϕ is defined as follows:

$$h_{\phi,g}(s) = d_{\phi(S)}(\phi(s), \phi(g))$$

where $d_X(s_1, s_2)$ is the distance – the length of the shortest path – from s_1 to s_2 in state space X.

It is important to realize that the definition of $\phi(S)$ does not guarantee that $\phi(S)$ is exactly equal to the image of S under ϕ. The definition guarantees that the image of S under ϕ is embedded in $\phi(S)$, but it also permits there to be states and connectivity in $\phi(S)$ that have no counterpart in S.

Definition 5 *Pre-image of an Abstract State*

The pre-image (under domain abstraction ϕ) of an abstract state $s \in \phi(S)$ is the set of states $\{s_0, s_1, ...s_n\}$ in S such that $\phi(s_i) = s$. If there are no such states the abstract state s is said to have no pre-image.

For example, the pre-image of $< 1, 1, 1, 0 >$ under ϕ_1 is the set of states $\{< 1, 2, 3, 0 >, < 2, 3, 1, 0 >, < 3, 1, 2, 0 >\}$.

The heuristic defined by an abstraction can either be computed on demand, as is done in Hierarchical A* [14], 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.

Definition 6 *Pattern Database*

Let $g \in S$ be a goal state and ϕ be an abstraction on S that induces a state space homomorphism. A pattern database is a table indexed by the states of $\phi(S)$. The entry in the table for $s' \in \phi(S)$ is $d_{\phi(S)}(s', \phi(g))$.

The size of the pattern database is defined to be the number of states in $\phi(S)$.

For a given state $s \in S$, $h_{\phi,g}(s)$ is computed by looking up the entry in the pattern database indexed by $\phi(s)$.

For example, the pattern database for the goal state $< 1,2,3,0 >$ of $S =$ the *2 × 2* sliding-tile puzzle and the abstraction $\phi_1(S)$ (see Figure 4) is shown in Table 1. $h(< 0,3,2,1 >)$ is computed by looking up the entry indexed by $\phi(< 0,3,2,1 >) =< 0,1,1,1 >$. This estimates the distance from $< 0,3,2,1 >$ to to $< 1,2,3,0 >$ to be 2 (the actual distance is 6).

s'	$d_{\phi_1(S)}(s', < 1,1,1,0 >)$
$< 1,1,1,0 >$	0
$< 1,1,0,1 >$	1
$< 1,0,1,1 >$	1
$< 0,1,1,1 >$	2

Table 1
Pattern database for ϕ_1

If S's successor function is symmetric, as we are assuming in this paper, the pattern database can be constructed by an exhaustive breadth first traversal of $\phi(S)$ starting at the goal state, $\phi(g)$, using the abstracted successor function.

The "patterns" in previous pattern database work are a restricted form of domain abstraction in which the labels that do not remain unique are all mapped to the same abstract label. ϕ_1 is a pattern in this sense. The granularity of these patterns thus consists of a single number (the rest of the entries being one) which simply indicates how many labels do not remain unique.

$GRAN^{\phi}$	m	n	$GRAN^{\phi}$	m	n
$< 8 >$	9	1	$< 4,2 >$	7560	420
$< 7 >$	72	8	$< 3,3 >$	10080	280
$< 6,2 >$	252	28	$< 3,2,2 >$	15120	210
$< 6 >$	504	28	$< 4 >$	15120	70
$< 5,3 >$	504	56	$< 2,2,2,2 >$	22680	105
$< 4,4 >$	630	35	$< 3,2 >$	30240	560
$< 5,2 >$	1512	168	$< 2,2,2 >$	45360	420
$< 4,3 >$	2520	280	$< 3 >$	60480	56
$< 5 >$	3024	56	$< 2,2 >$	90720	210
$< 4,2,2 >$	3780	210	$< 2 >$	181440	28
$< 3,3,2 >$	5040	280	$< 1,1,1,1,1,1,1,1,1 >$	181440	1

Table 2
Abstraction granularities for the 8-puzzle.

To see the advantage of the richer notion of domain abstraction, consider Table 2, which shows all the possible granularities of domain abstractions of the 8-puzzle in which the blank tile remains unique in the abstract space. n is the number of distinct domain abstractions that have a given granularity, and m is the number of abstract states generated by each such abstraction. For example, the entry for $< 3,3,2 >$ indicates that there are $n = 280$ different domain abstractions with this granularity, each of which produces an abstract space containing $m = 5040$ states. Note that different granularities sometimes produce the same number of abstract states (e.g. $< 3,2,2 >$ and $< 4 >$).

The entries in this table that correspond to "patterns" are the single-digit granularities – $< 8 >, < 7 >, < 6 >$ and so on. Note that in between successive patterns from $< 6 >$ to $< 3 >$ the size jumps considerably (relatively speaking). The gap between the sizes becomes important when one size is much smaller than the amount of memory available while the next size is bigger than the memory available. In [19] for example, a pattern in which 6 of the Rubik's Cube corner cubies remain unique produces a 20 megabyte pattern database, while the next larger pattern produces a 244 megabyte pattern database. If the actual memory available had been 200 megabytes the larger pattern database could not have been used but the smaller one would have used only 10% of the available memory. It is important to fully utilize memory because larger pattern databases usually result in faster search (see below).

The domain abstractions that are not patterns fill in these gaps. In Table 2 between $< 6 >$ and $< 5 >$, and again between $< 5 >$ and $< 4 >$ there are a variety of intermediate sizes.

3 Efficiently Identifying Good Pattern Database Heuristics

Generally speaking, search time is inversely related to pattern database size[13,19]. This is illustrated in Figure 5, which shows how the average search performance on 400 start states of all the pattern databases in Table 2 grouped together by size. Clearly, the number of nodes expanded decreases as the size of the pattern database increases.

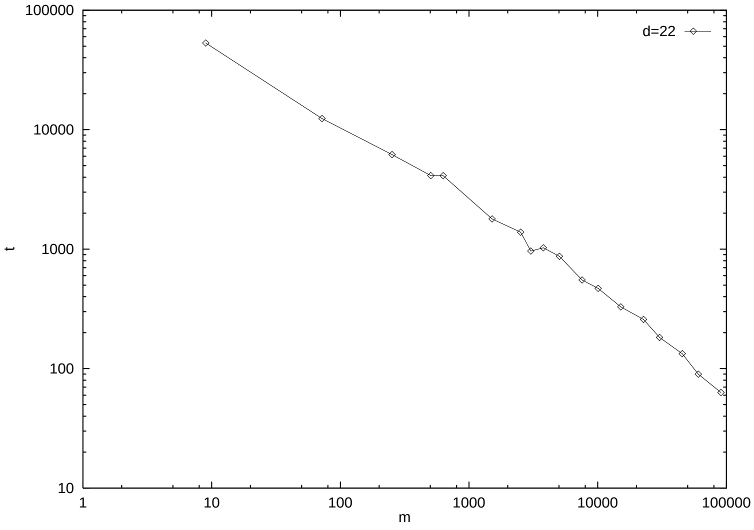


Fig. 5. 8-Puzzle: Number of Nodes Expanded (y axis) *vs* Pattern Database Size (x axis). Logarithmic scales on both axes.

However, pattern database size is not a perfect predictor of search performance. Figure 6 shows the same data, but with each pattern database plotted individually. Here it can been seen that there is significant overlap from one size to the next: the best heuristic with less memory is often better than an average heuristic with slightly more memory.

This observation is the primary motivation for the current section, which investigates a particular formula for predicting the search performance of a heuristic. Our interest in this formula is to see how well it serves as the scoring function in the search procedure in Figure 1 to select the "best" pattern database of size m or less. Note that, for this purpose, it is not essential to predict true performance exactly, it is only necessary to correctly determine which of two pattern databases defines the better heuristic.

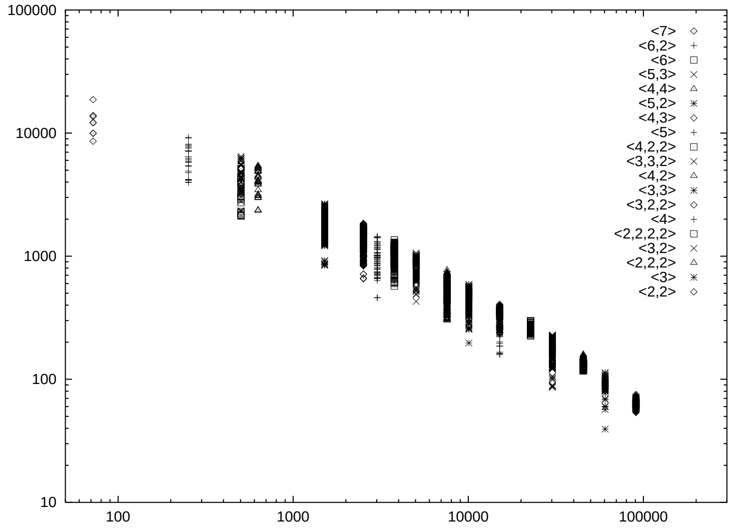


Fig. 6. Number of Nodes Expanded by Individual Pattern Databases (y axis) *vs.* Pattern Database Size (x axis). Logarithmic scales on both axes. The legend shows granularity in order of increasing size.

3.1 Korf and Reid's formula predicting the number of nodes expanded

In [20] Korf and Reid develop a formula to predict the number of nodes generated during search as a function of parameters that can be easily estimated for pattern databases. Our reconstruction of their development, with some slight differences, is as follows.

Let S_d be the set of states distance d from the goal, let $Nodes(s,i,d)$ be the set of nodes at level i of the search tree when $s \in S_d$ is the start state, and let $Fertile(s,i,d)$ be the set of $x \in Nodes(s,i,d)$ such that $f(x) = g(x) + h(x) = i + h(x) \leq d$. Korf and Reid's key insight is that the total number of nodes expanded for a search from $s \in S_d$ to the goal is at most

$$t(s,d) = \sum_{i=0}^{d} |Fertile(s,i,d)| \tag{1}$$

The expected number of nodes when starting distance d from the goal is then

$$t(d) = \frac{\sum_{s \in S_d} \sum_{i=0}^{d} |Fertile(s,i,d)|}{|S_d|} \tag{2}$$

Defining $P(s,i,d) = |Fertile(s,i,d)|/|Nodes(s,i,d)|$ this equation can be rewritten

$$t(d) = \frac{\sum_{s \in S_d} \sum_{i=0}^{d} |Nodes(s,i,d)| \cdot P(s,i,d)}{|S_d|} \tag{3}$$

The condition defining $Fertile$ is equivalent to $h(x) \leq d - i$, and therefore $P(s, i, d)$ is the probability of encountering a node with a heuristic value less than or equal to $d - i$ at depth i of the search tree with $s \in S_d$ as its starting node.

Assuming $|Nodes(s, i, d)|$ is the same for all $s \in S_d$ the equation can be rearranged as

$$t(d) = \sum_{i=0}^{d} \left(|Nodes(i, d)| \cdot \frac{\sum_{s \in S_d} P(s, i, d)}{|S_d|} \right) \tag{4}$$

The latter part of the equation is just the average value of $P(s, i, d)$ for $s \in S_d$. Denote this by $\bar{P}(i, d)$.

$$t(d) = \sum_{i=0}^{d} |Nodes(i, d)| \cdot \bar{P}(i, d) \tag{5}$$

The development to this point has used exact definitions for $|Nodes(i, d)|$ and $\bar{P}(i, d)$. Unfortunately, in general it is not possible to efficiently compute either of these values. To complete the development, easily computed approximations of $|Nodes(i, d)|$ and $\bar{P}(i, d)$ must be introduced, and this is where complications arise.

$|Nodes(i, d)|$ is usually approximated as b^i by assuming a constant branching factor, b, across the entire space and strict exponential growth of the search tree, but we have delayed this substitution until now to show that the analysis does not depend on it in any way. Faced with a specific search space about which one has knowledge pertaining to $|Nodes(i, d)|$ a more appropriate approximation could be used. In addition, the search algorithm must be taken into account. The assumption of a constant branching factor is reasonable when the IDA* search algorithm[17] is being used, although calculating the correct value for the branching factor is trickier than it might at first seem[7]. By contrast, the branching factor is certainly not constant when the A* search algorithm[10] is being used. Because A* detects and prunes duplicate nodes, its branching factor is smaller at deeper levels.

An easily computable approximation of $\bar{P}(i, d)$, with heuristics defined by pattern databases, is $\tilde{P}(x)$, which is defined to be the percentage of the entries in the pattern database that are less than or equal to x. $\tilde{P}$ is the cumulative distribution of the heuristic values stored in the pattern database, which is related to, but different than, the overall distribution used in [20]. However, if each abstract state has the same number of pre-images, then $\tilde{P}$ and the overall distribution are the same. This is the case for the abstractions in our experiments, but it is not true in general (see sections 4 and 5), and that is the key difficulty in trying to approximate $\bar{P}(i,d)$ from a pattern database. A second difficulty in using this approximation arises with the IDA* search algorithm. Because IDA* prunes nodes with sufficiently large heuristic values but it does not detect and prune duplicate nodes, nodes with small heuristic values will tend to recur more often than those with large heuristic values (a similar observation is made in [19]). There is no efficiently computable way to accurately adjust $\tilde{P}(x)$ to account for this effect. It is primarily for this reason that we have used A* instead of IDA* in our experiments.

Substituting these two approximations into the equation for $t(d)$ produces the the final formula we will use for estimating the number of nodes expanded by a search starting distance d from the goal in a space with branching factor b

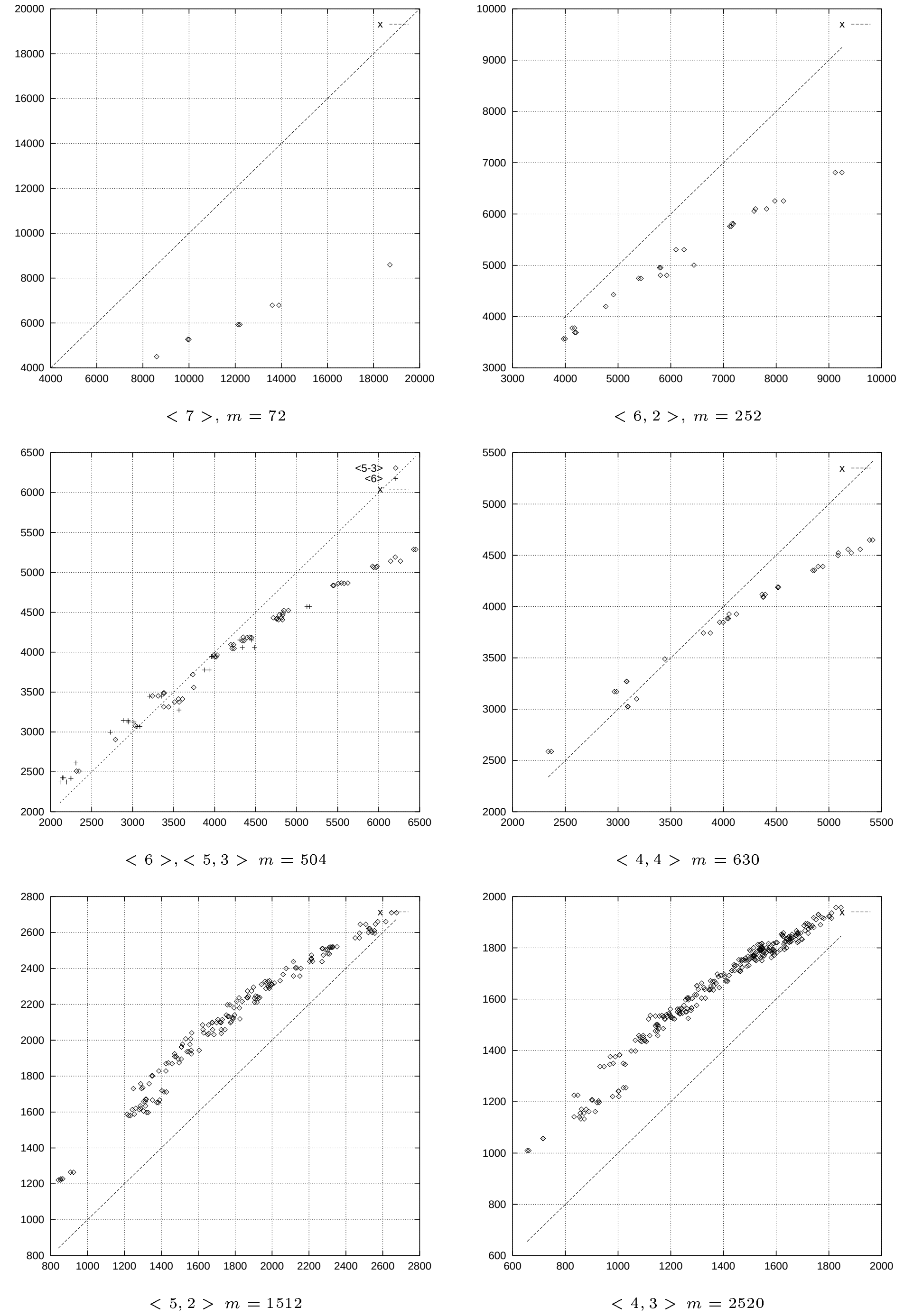
$$t(b,d) = \sum_{i=0}^{d} b^i \tilde{P}(d-i) \qquad (6)$$

3.2 Experimental Evaluation assuming b and d are known

To verify how well equation 6 predicts the number of nodes expanded we have run an extensive experiment with the 8-Puzzle. We generated all 3510 different abstractions described in Table 2 and evaluated them on the same 400 start states randomly chosen from among the 23952 states that are distance 22 from our chosen goal state. The experiment thus involved solving 1,404,000 problem instances. The 8-puzzle was used because it is large enough to be interesting but small enough that such a large-scale experiment was feasible.

Each point in Figures 7 - 8 represents the average number of nodes expanded on the 400 start states for one particular pattern database. The granularity and the size of the pattern database are shown under each figure. Where there are two different granularities that produce the same size pattern databases (e.g. $< 6 >$ and $< 5,3 >$ both produce 504 abstract states) they are shown on the same plot with different symbols.

Figures 7 and 8 show the predicted number of nodes expanded on the y axis versus the actual values on the x axis when b and d are both known. If equation 6 precisely predicted the number of nodes expanded, the points would lie on the $y = x$ line. The points representing the small memory heuristics ($m = 72$ to $m = 630$) fall below the $y = x$ line indicating that equation 6 underestimates the number of nodes expanded even though this equation was derived to be an upper bound. This anomaly is because b^i is not a perfect estimate of $Nodes(i,d)$. In particular, the use of the same branching factor, b, at each level is problematic for inaccurate heuristics. As discussed above, A*'s duplicate-node pruning causes the effective branching factor to decrease as A* reaches deeper levels. The value for b we used in equation 6 is the arithmetic average



$< 7 >,\ m = 72$

$< 6, 2 >,\ m = 252$

$< 6 >, < 5, 3 >\ \ m = 504$

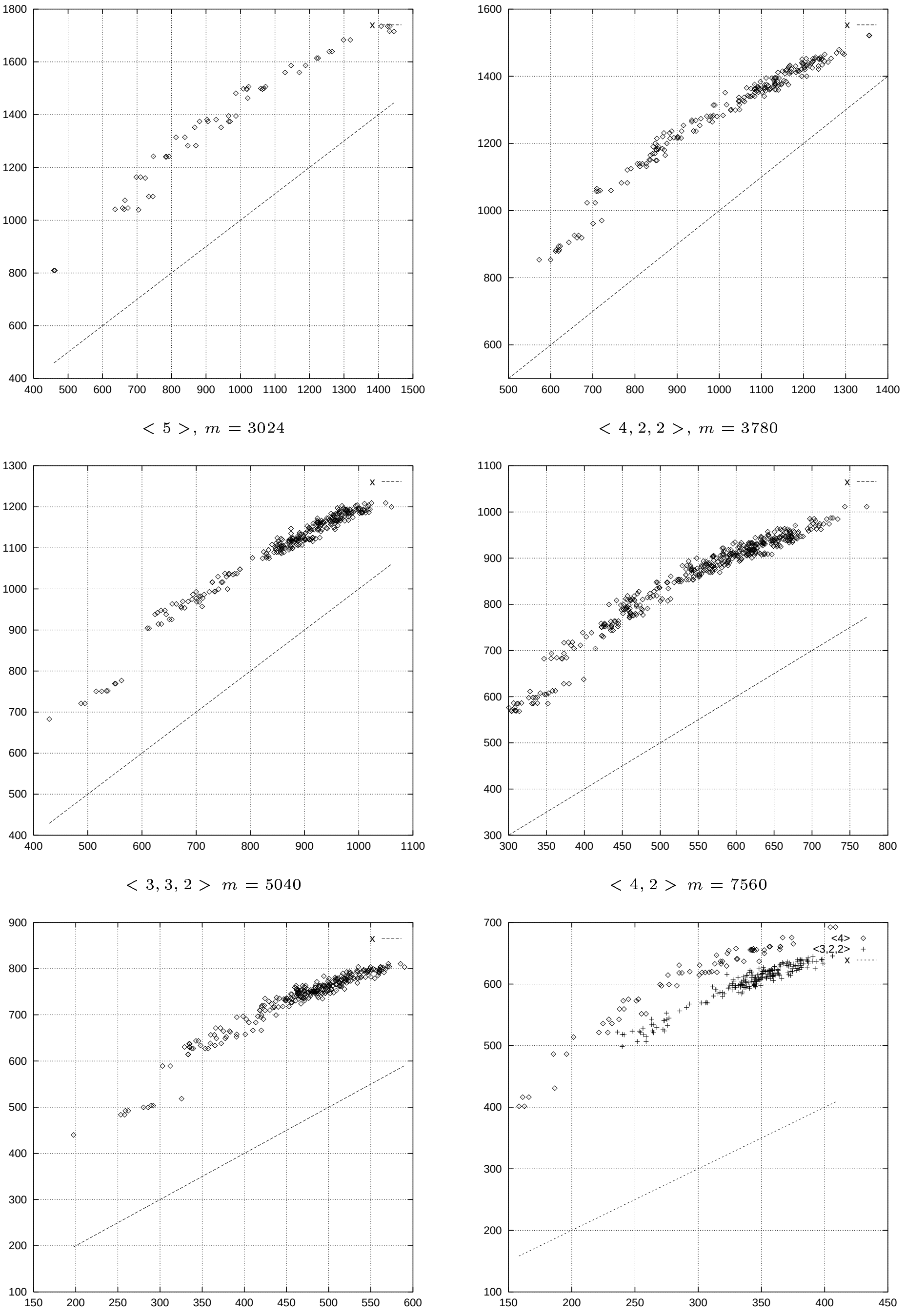
$< 4, 4 >\ \ m = 630$

$< 5, 2 >\ \ m = 1512$

$< 4, 3 >\ \ m = 2520$

Fig. 7. Number of Nodes Expanded as Predicted by Equation 6 (y axis) *vs.* The Average of the Actual Number of Nodes Expanded (x axis)

of the effective branching factor of every node that was expanded. This underestimates b for small depths and overestimates it for large depths. However, equation 6 weighs b^i by $\tilde{P}(d-i)$ where i is the search depth. Since $\tilde{P}(d-i)$ is greatest for the small depths, and b is significantly underestimated at these depths for small memory heuristics, the overall formula underestimates the total number of nodes expanded.



$< 5 >$, $m = 3024$

$< 4, 2, 2 >$, $m = 3780$

$< 3, 3, 2 >$ $m = 5040$

$< 4, 2 >$ $m = 7560$

$< 3, 3 >$ $m = 10080$

$< 4 >$, $< 3, 2, 2 >$ $m = 15120$

Fig. 8. Number of Nodes Expanded as Predicted by Equation 6 (y axis) *vs.* The Average of the Actual Number of Nodes Expanded (x axis)

Nevertheless, it is clear that for all sizes of heuristic the points in Figures 7 - 8 form a pattern that is sufficiently monotonic, on a large scale, that equation 6 can be used to reliably determine which of two heuristics will result in fewer nodes being expanded.

3.3 Using Korf and Reid's formula when b and d are not known

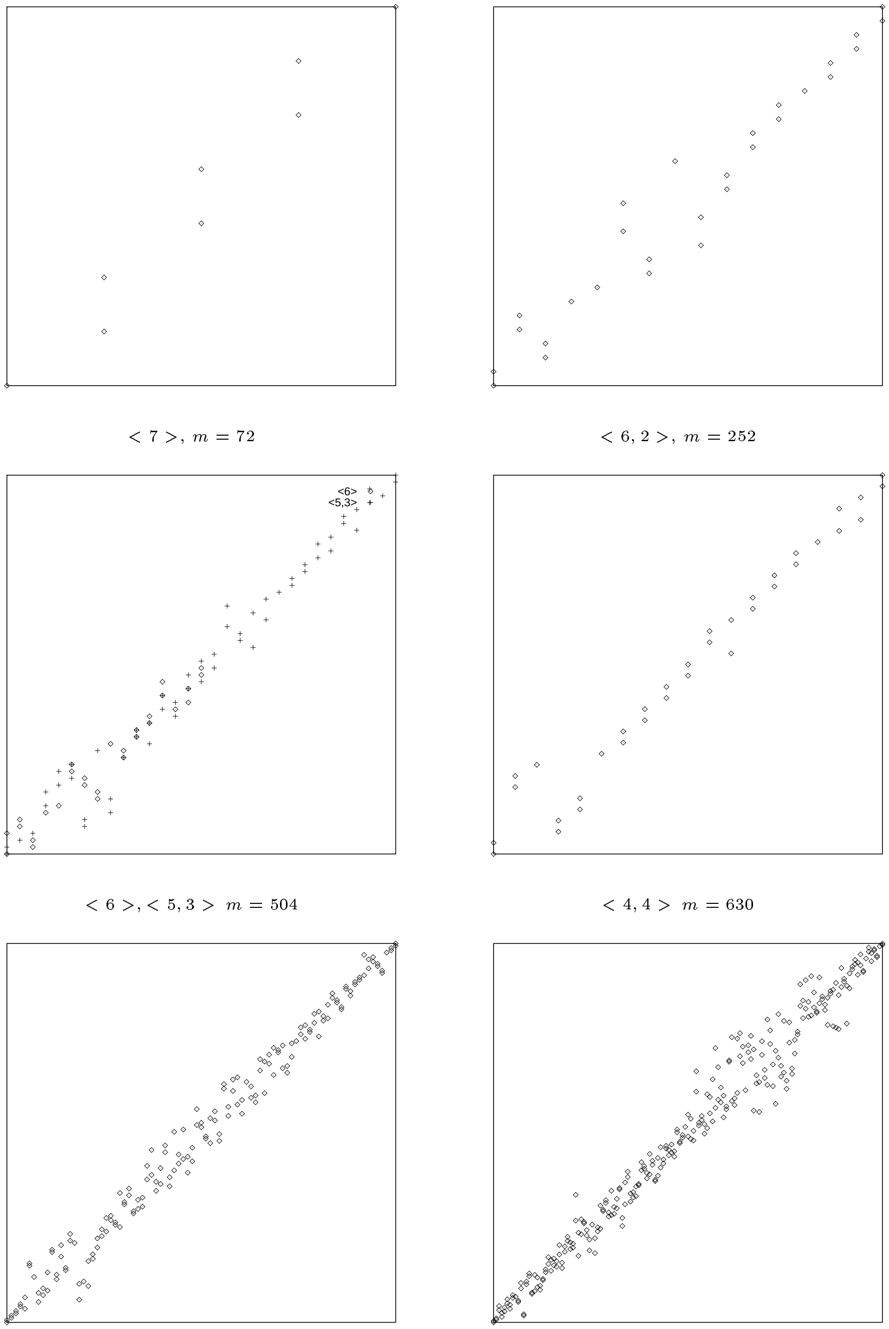
We have seen that equation 6 (Figures 7 - 8) accurately orders the heuristics with respect to the number of states they expand when b and d are known exactly. However, these values are unknown at the time we wish to compare heuristics. b can be estimated by probing the search space, i.e., by searching forward to a limited depth from a number of randomly chosen start states and measuring the average branching factor in those searches.

Unlike b, d is not a fixed quantity to be estimated, since our aim is to determine which of two heuristics will result in fewer nodes expanded over a set of unknown start states. Assuming the number of nodes expanded grows exponentially in d, the better heuristic is the one with the smaller $t(b, d)$ for large values of d. Let h denote the largest heuristic value stored in a given pattern database. For $x > h$, $\tilde{P}(x) = 1$, so $t(b, x)$ for $x > h$ is monotone non-decreasing for $b > 1$. Hence given two pattern databases with largest heuristic values h_1 and h_2, respectively, the one with the smaller $t(b, max(h_1, h_2))$ will expand fewer states when searching to large depths. To select the best among n pattern databases with the same storage requirements, we choose the one with the smallest $t(b, max(h_1, h_2, ..., h_n))$.

In Figures 9 - 10 the actual rank with respect to the experimentally measured the number of nodes expanded is on the x axis and the rank established by equation 6 is on the y axis.

If equation 6 ranked heuristics perfectly then the plots on Figures 9 - 10 would be diagonals from the bottom-left to the top-right corner. It is clear that the formula ranks quite accurately and, in particular, that its best-ranked heuristics are always very good, if not best, in their actual performance.

Figure 11 compares the performance of the best ranked, the average of the best ranked 10 and the average of all pattern databases on the same 400 start states. The actual values are also tabulated in Table 3. The best heuristics are two to three times faster than the average ones with the same size. This also means that the best pattern database of size m often performs as well or better than an average pattern database of size $2m$. For $d = 22$, which has been fixed throughout our experiments, the Manhattan distance heuristic expands 975 nodes, on average. From Table 3 the average of the pattern databases of size 3024 would perform slightly better, as would the best pattern database of size 1512.



$< 7 >$, $m = 72$

$< 6, 2 >$, $m = 252$

$< 6 >, < 5, 3 >$ $m = 504$

$< 4, 4 >$ $m = 630$

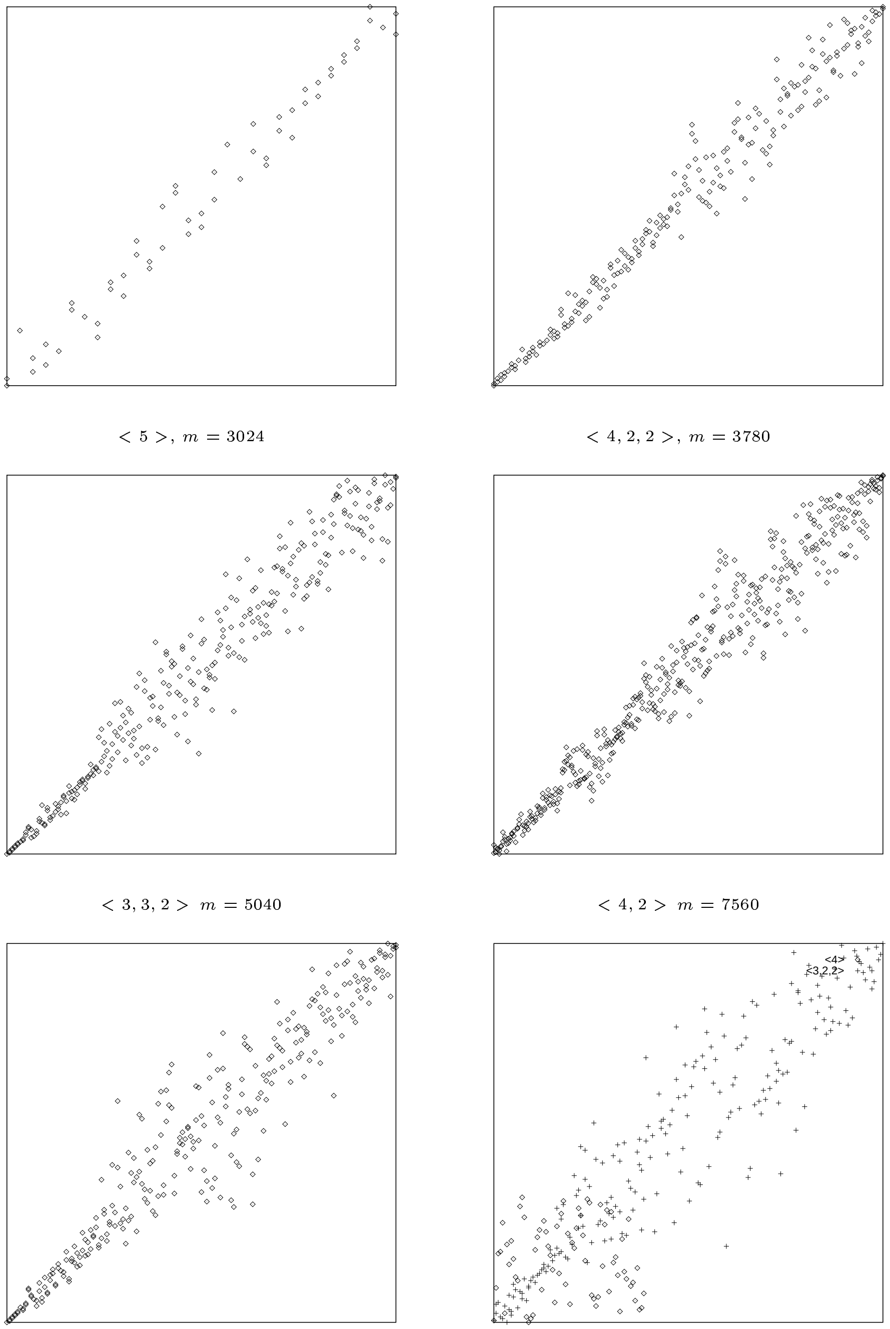
$< 5, 2 >$ $m = 1512$

$< 4, 3 >$ $m = 2520$

Fig. 9. Rank as Determined by Equation 6 (y axis) *vs.* Actual Rank (x axis) (The granularity and the size of the pattern database are placed under each plot).

4 Creating Pattern Databases of a Given Size

Given a state space and a maximum pattern database size, m, one would like an automatic method for generating abstractions that give rise to pattern databases of size m or less.



$< 5 >$, $m = 3024$

$< 4, 2, 2 >$, $m = 3780$

$< 3, 3, 2 >$ $m = 5040$

$< 4, 2 >$ $m = 7560$

$< 3, 3 >$ $m = 10080$

$< 4 >$, $< 3, 2, 2 >$ $m = 15120$

Fig. 10. Rank as Determined by Equation 6 (y axis) *vs.* Actual Rank (x axis) (The granularity and the size of the pattern database are placed under each plot).

For special classes of state spaces there exists a simple formula relating the definition of an abstraction to the size of the corresponding pattern database. For example, consider the N-Perm puzzle. In this space a state is a permutation of the values $0...(N-1)$. A state has $N-1$ successors, with the k^{th} successor formed by reversing the order of the first $k+1$ elements of the permutation ($1 \leq k < N$). For example, if $N = 5$ the successors of state $< 0, 1, 2, 3, 4 >$

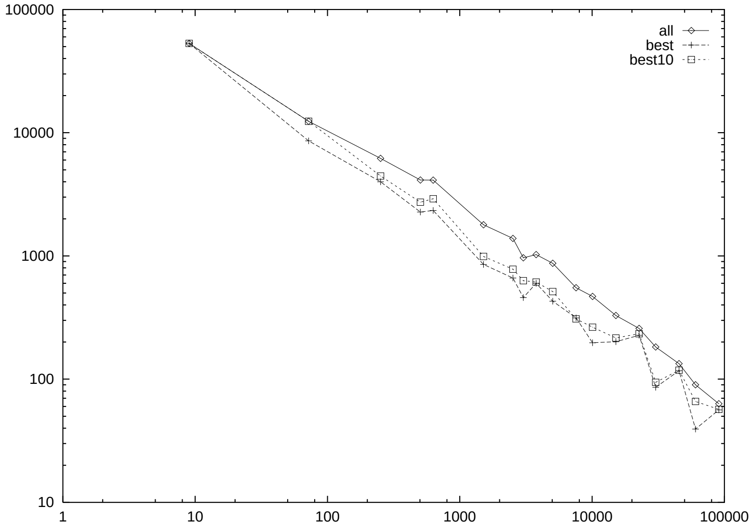


Fig. 11. Size of Pattern Database (x axis) *vs.* Number of Nodes Expanded (y axis) by All, the Best and the Best 10 Pattern Databases of Each Granularity. Logarithmic scales on both axes.

are $< 1, 0, 2, 3, 4 >, < 2, 1, 0, 3, 4 >, < 3, 2, 1, 0, 4 >$ and $< 4, 3, 2, 1, 0 >$. From any state it is possible to reach any other permutation, so the size of the state space, $|S|$, is $N!$.

In the natural encoding of this space, the domain has N elements representing the N distinct values that are being permuted. Suppose an abstraction maps this domain to domain D' having M values $(M < N)$, with g_i of the original domain elements mapped to D'_i $(0 \leq i \leq M - 1)$. For such an abstraction the number of states in the original space which map to each abstract state is

$$g_0!g_1! \cdot \ldots \cdot g_{(M-1)}!$$

and therefore the number of abstract states is

$$\frac{|S|}{g_0!g_1! \cdot \ldots \cdot g_{(M-1)}!}$$

Unfortunately, an analogous formula cannot be used in general, for two reasons. One reason is that the abstract space may contain abstract states that have no pre-image, and the formula just described makes no allowance for these. This phenomenon is discussed in detail in Section 5.

The second reason is that the fundamental assumption underlying the formula – that the number of states mapped to each abstract state is the same – is not true in general, even if abstract states with no pre-image are ignored. For example, consider the 3-disk Towers of Hanoi puzzle with a domain, $\{1, 2, 3\}$, whose labels represent the different disks (1=small, 2=medium, 3=big), and

database		**nodes expanded**		
m	$GRAN^{\phi}$	**best**	**best 10**	**all**
9	$< 8 >$	53097	53097	53097
72	$< 7 >$	8597	12384	12384
252	$< 6, 2 >$	3998	4460	6192
504	$< 5, 3 >$	2346	3104	4515
504	$< 6 >$	2195	2355	3356
630	$< 4, 4 >$	2338	2909	4123
1512	$< 5, 2 >$	854	991	1791
2520	$< 4, 3 >$	662	780	1386
3024	$< 5 >$	459	630	965
3780	$< 4, 2, 2 >$	600	614	1026
5040	$< 3, 3, 2 >$	429	514	872
7560	$< 4, 2 >$	314	308	552
10080	$< 3, 3 >$	197	264	469
15120	$< 4 >$	163	182	293
15120	$< 3, 2, 2 >$	240	250	341
22680	$< 2, 2, 2, 2 >$	227	234	258
30240	$< 3, 2 >$	86	94	182
45360	$< 2, 2, 2 >$	117	118	134
60480	$< 3 >$	40	66	90
90720	$< 2, 2 >$	56	57	63

Table 3
Performance of the the Best, the Best 10 and all pattern databases on the same 400 start states

the abstraction that maps all these labels to the same abstract label (say, X). With this abstraction there are still three disks but their sizes are indistinguishable. The abstract state in which the disks are all on different pegs has a pre-image containing six Towers of Hanoi states, since any way of replacing the three X's by distinct labels ($\{1, 2, 3\}$) is a legal, reachable Towers of Hanoi state. But the abstract state in which the disks are all on the same peg has only one Towers of Hanoi state in its pre-image, since only one legal, reachable Towers of Hanoi state is mapped to this abstract state (1 on top, 2 in the middle, 3 on the bottom).

One consequence of pre-images being different sizes is that it is possible for two domain abstractions of the same granularity to produce abstract spaces with a different number of states, even if all the abstract states have non-empty pre-images.

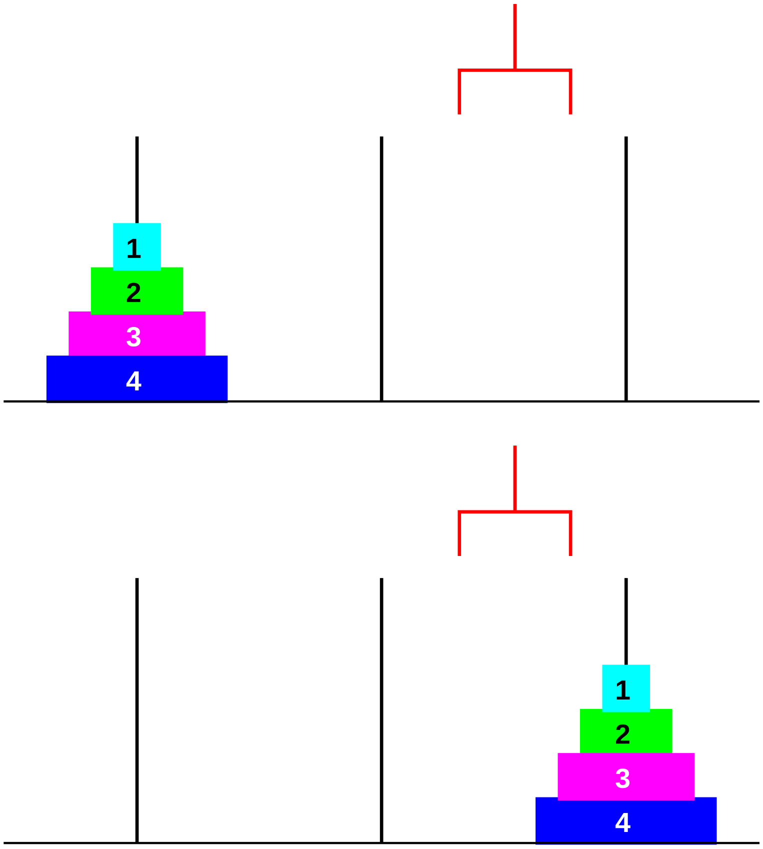


Fig. 12. Towers of Hanoi with four disks and a hand

To illustrate this consider the variation of the Towers of Hanoi puzzle depicted in Figure 12 in which the operations are akin to those in the Blocks World: there is a "robot hand" which, if it is empty, can pick up the disk on top of any peg and which can put down the disk it is holding on any empty peg or on top of a larger disk atop a peg. Suppose there are 4 disks represented by labels $\{1, 2, 3, 4\}$ (1=smallest, ..., 4=biggest), and consider the following abstractions

$$\phi_{small}(1) = \phi_{small}(2) = a$$

$$\phi_{small}(3) = b$$

$$\phi_{small}(4) = c$$

$$\phi_{big}(1) = a$$
$$\phi_{big}(2) = b$$
$$\phi_{big}(3) = \phi_{big}(2) = c$$

ϕ_{small} maps the two smaller disks to the same abstract label and leaves the larger disks unique, while ϕ_{big} does the opposite, mapping the two larger disks to the same abstract label and leaving the smaller disks unique. Both have granularity $< 2, 1, 1 >$, and both create abstract spaces in which all the abstract states have non-empty pre-images. But the abstract space produced by ϕ_{big} has three abstract states more than the space produced by ϕ_{small} (117 compared to 114). The difference increases as the number of disks is increased. With 7 disks, the abstraction mapping the 3 largest disks to the same abstract label and leaving the smaller disks unique produces an abstract space with 144 states more than the abstraction that maps the 3 smallest disks to the same abstract label and leaves the larger disks unique (2100 compared to 1956).

4.1 Searching the Space of Domain Abstractions

Since it is not always possible to predict the size of the space induced by a domain abstraction, it is necessary to search in the space of domain abstractions to find ones that produce pattern databases of a desired size. Because domains are finite (indeed usually very small) it is easy to represent and enumerate all possible abstractions of domain D. An abstraction ϕ can be represented by a vector of length $|D|$ with the value in position i representing $\phi(D_i)$.

Any type of search algorithm could be used for this purpose; a typical local search algorithm is given in Figure 1. The search considers one abstraction at a time. If the abstract space induced by the current abstraction is larger than desired, the abstraction is changed to be less abstract. On the other hand, if the induced space is smaller than desired, the abstraction is changed to be more abstract.

A method to change the degree of abstractness (steps 2(b) and 2(c)(iii) in Figure 1) can be based on the effect of changing the g_i values on the formula

$$\frac{|S|}{g_0!g_1! \cdot \ldots \cdot g_{(|D|-1)}!}$$

If a label that is currently mapped to D'_i is changed to be mapped to D'_j the previous value of the formula will be multiplied by

$$\frac{g_i}{g_j + 1}$$

because g_i will decrease by 1 and g_j will increase by 1. If $g_i \leq g_j$ the overall value of the formula will decrease, and if $g_i > (g_j + 1)$ the value will increase. Therefore, to make a given abstraction "more abstract" one can re-assign any label mapped to D'_i to any D'_j such that $g_i \leq g_j$, and to make an abstraction "less abstract" one can re-assign any label mapped to D'_i to any D'_j such that $g_i > (g_j + 1)$. This method is only a heuristic: it is not guaranteed to achieve the expected effects because, for the reasons given above, the formula on which it is based does not always exactly predict the size of the abstract space.

As an initial abstraction to begin the search (step (1) in Figure 1) two extreme possibilities suggest themselves: the maximum abstraction, which maps the whole original domain to the same abstract label, creating an abstract space with just one state, and the minimum abstraction, which maps each label in D to a distinct abstract label, creating an abstract space that is isomorphic to the original space.[3] Because the changes made to an abstraction typically have a multiplicative effect on the size of the abstract space even with these extreme starting points the search is expected to quickly find abstractions that are roughly size m.

Figure 1 does not include any loop exit conditions because these are chosen to suit the particular application. For instance, in some cases the loop might exit as soon as the first pattern database smaller than m is generated, whereas in others the search might continue until a sufficiently high score had been achieved. Yet other applications might continue until a pre-determined number of iterations had been executed.

5 Non-Surjective Abstractions

The main complication in trying to predict or control the size of the pattern database induced by a domain abstraction is that an abstract space can contain an arbitrarily large number of states that have no pre-images. We call such an abstraction *non-surjective.*

Definition 7 *Surjective State-Space Abstraction*

[3] this is not an "abstraction" according to Definition 1 but it is a convenient initial value for ϕ in the search procedure being discussed.

Let ϕ be a domain abstraction and $\phi(S)$ be the abstract space induced by applying ϕ to state space S. ϕ is surjective if and only if for every state $s' \in \phi(S)$ there is a state $s \in S$ such that $s' = \phi(s)$.

In other words, ϕ is non-surjective if $\phi(S)$ contains states with no pre-image.

Non-surjective abstractions arise quite often, for a variety of reasons. Detecting that an abstract state has no pre-image requires solving to a more general reachability problem, which is not necessarily efficiently computable. Alternatively, one might attempt to characterize the causes of non-surjectivity, and then develop automatic methods to detect or avoid non-surjective abstractions.

To date, we have identified two major causes of non-surjectivity, *orbits* (Section 5.4) and *blocks* (Section 5.5). 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.

Unfortunately, these are not the only causes. Sections 5.1 to 5.3 give examples of non-surjective abstractions which are not due to orbits or blocks.

5.1 Disk Order Not Preserved in the Towers of Hanoi

Consider the 4-disk Towers of Hanoi puzzle and the domain abstraction:

$$\phi(1) = \phi(4) = a$$
$$\phi(2) = b$$
$$\phi(3) = c$$

ϕ assigns the same abstract label to the smallest and largest disks and leaves the two intermediate disks unique. In the original space the largest disk can never be on top of another disk and the smallest can never be beneath another disk. These restrictions cannot be enforced in the abstract space. The largest and smallest disks are indistinguishable from each other in the abstract space so and they get mixed in with the rest of the disks. Consequently, many abstract states have no pre-images. For example, "b on a on a on c on peg1" is a reachable abstract state, which obviously has no pre-image. This problem arises in the original Towers of Hanoi puzzle and in the "robot hand" variation (Figure 12). In the latter, $\phi(S)$ actually has more states than the original space (258 abstract states compared to 183 original states).

5.2 *Multiple blanks in the 2 × 2 puzzle*

The abstraction, ϕ_1, of S = the *2 × 2* puzzle discussed in Section 2.2 is surjective: each state of $\phi_1(S)$ has a (non-empty) pre-image. But consider the domain abstraction ϕ_2 defined as:

$$\phi_2(0) = \phi_2(3) = 0$$
$$\phi_2(1) = 1$$
$$\phi_2(2) = 2$$

ϕ_2 creates two blank tiles (0's) in the abstract space. The 12 original reachable states are mapped to 8 abstract states[4] : this is the size of the "image" of the initial state space under ϕ_2. However, the abstract space, shown in Figure 13, has 12 states in total. The abstract states in the image of the *2 × 2* puzzle have solid boundaries; the 4 reachable abstract states that do not correspond to reachable *2 × 2* puzzle states have dashed boundaries. Because there are states in $\phi_2(S)$ which have no pre-image, ϕ_2 is non-surjective.

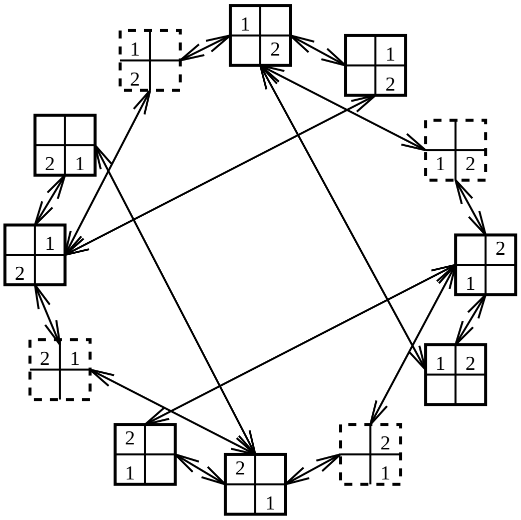


Fig. 13. Abstract space induced by $\phi_2(S)$

[4] the details in this paragraph depend on the encoding of the problem. Here we are assuming the encoding used is the one given above. In the next subsection we will see a second encoding in which the image has 9 states.

5.3 Dual encoding of the 2 × 2 puzzle

In this section we describe an alternative way of representing the *2* × *2* puzzle which we call the dual representation. This representation is quite natural, but has the unfortunate property that all non-trivial domain abstractions of it are non-surjective.

In the representation of the *2* × *2* puzzle used so far, a state was represented by vector of length four with the vector positions corresponding to the positions in the puzzle, and the four domain labels representing the different tiles. An alternative is to have each vector position correspond to one of the tiles, with the label in that position representing which puzzle position that tile is currently in. For example, let vector positions 1-3 correspond to tiles 1-3 respectively and position 4 correspond to the blank tile. In this representation there will be four labels, one for each puzzle position. The labels are $\{tl, tr, bl, br\}$, where tl stands for “top left” tr for “top right” and so on.

Examples of this *dual* representation are shown in Figure 14. Vector $< br, tl, tr, bl >$ represents the state where tile 1 is in the bottom right position, tile 2 is in the top left, tile 3 is in the top right and the blank tile is in the bottom left.

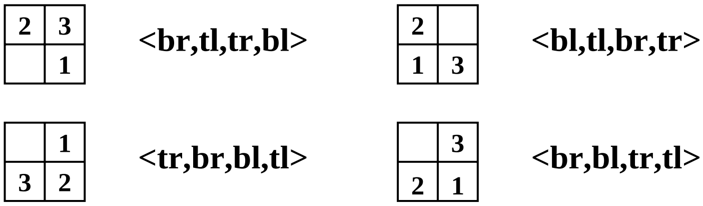


Fig. 14. Examples of States in the Dual Representation of the *2* × *2* Puzzle

In this representation, a domain abstraction abstracts the positions the tiles occupy rather than the tiles themselves and renders certain positions indistinguishable from one another. This effectively allows the tiles in those positions to be swapped, which almost always leads to a state with no pre-image. For example, consider domain abstraction ϕ_4 defined as

$$\phi_4(tr) = \phi_4(br) = br$$

$$\phi_4(tl) = tl$$

$$\phi_4(bl) = bl$$

This abstraction renders positions tr and br indistinguishable from one another. Figure 15 shows the abstract space. The abstract states of $\phi_4(S)$ are drawn as solid or dashed ellipses each containing 2 puzzle states. Reachable puzzle states are drawn with solid boundaries, and those that are not reachable are drawn with dashed boundaries. In total $\phi_4(S)$ has 12 states; the 3 abstract states (ellipses) with dashed boundaries contain only unreachable puzzle states and therefore do not have a pre-image in the *2 × 2* Puzzle. Hence ϕ_4 is not surjective.

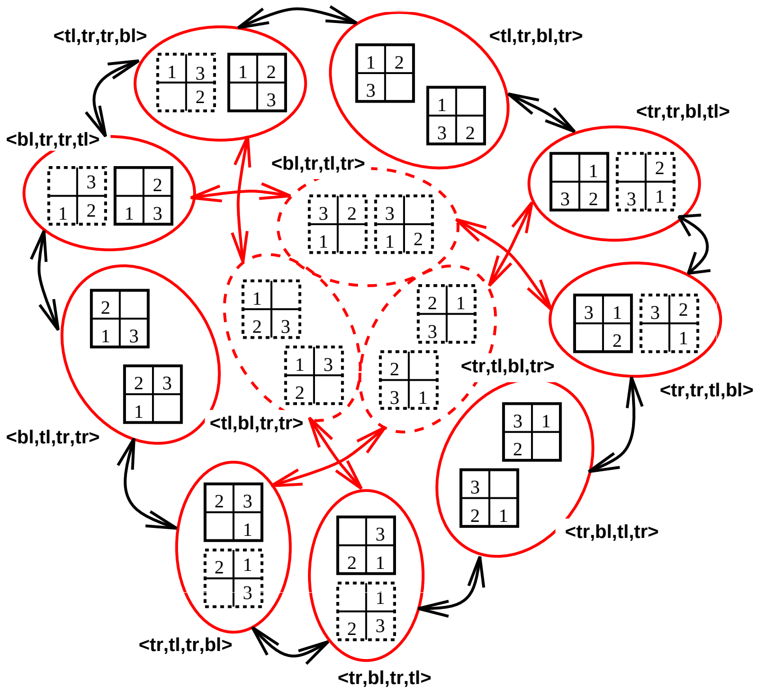


Fig. 15. $\phi_4(S)$

5.4 Orbits

The preceding subsections have given examples of non-surjective abstractions that are somewhat problem-specific, although we believe they are suggestive of general characteristics causing non-surjectivity. In this and the next subsection, we give two general causes of non-surjectivity, by generalizing the standard concepts of orbits and blocks found in Group Theory. Our adaptations assume states are represented by fixed-length vectors.

Definition 8 *Orbit*

Let x be a label and S a set of states represented as vectors. The orbit of x in S, denoted x^S, is the set of vector positions where x occurs in the states in S:

$$x^S = \{i : s \in S \text{ and } s_i = x\}$$

In the sliding tile puzzles, every tile can be moved to every puzzle position, so there is no interesting orbit structure: every label can occur in every vector position. By contrast, in Rubik's cube, there is a very important orbit structure (assuming labels represent the individual “cubies”): corner cubies can only be moved to the corner positions and edge cubies can only moved the edge positions. This is an example of a space in which the orbits of some labels have no overlap with the orbits of other labels. To see how this sort of orbit structure interacts with domain abstraction, consider the very simple space, S_1, with this PSVN definition:

$$seed =< a, b, c, d > \quad domain = \{a, b, c, d\}$$

$$succ = \left\{ \begin{array}{l} o_1 :< A, B, \_, \_ > \to < B, A, \_, \_ > \\ o_2 :< \_, \_, A, B > \to < \_, \_, B, A > \end{array} \right\}$$

S_1 has four states: $< a, b, c, d >$, $< a, b, d, c >$, $< b, a, c, d >$ and $< b, a, d, c >$. The orbits of labels a and b are the same, $\{1, 2\}$, and have no overlap with the orbits of c and d, which are both $\{3, 4\}$.

Consider the domain abstraction

$$\phi(a) = a$$
$$\phi(b) = \phi(c) = c$$
$$\phi(d) = d$$

$\phi(S_1)$ has four states: $< a, c, c, d >$, $< a, c, d, c >$, $< c, a, c, d >$ and $< c, a, d, c >$ and is isomorphic to S_1. Even though $\phi(S_1)$ appears to be a nontrivial abstraction – $GRAN^{\phi} =< 2 >$ – the abstract space is exactly the same as the original space. This is because labels b and c have non-overlapping orbits and therefore can be distinguished by their position even after they have both been changed to c.

Non-surjectivity arises when labels that have partially overlapping orbits are mapped to the same abstract label. The orbits become merged in the abstract space allowing the labels to move to positions they could not reach in the original space.

To illustrate this consider the state space, S_2, with this PSVN definition:

$$seed =< a, b, b, b, c > \quad domain = \{a, b, c\}$$

$$succ = \left\{ \begin{array}{l} o_1 :< A, B, b, \_, \_ > \rightarrow < b, A, B, \_, \_ > \\ o_2 :< A, B, a, \_, \_ > \rightarrow < a, A, B, \_, \_ > \\ o_3 :< \_, \_, c, A, B > \rightarrow < \_, \_, A, B, c > \\ o_4 :< \_, \_, b, A, B > \rightarrow < \_, \_, A, B, b > \end{array} \right\}$$

S_2 has the following states:

$$\begin{array}{c} < a, b, b, b, c >,\ < a, b, b, c, b >, \\ < b, a, b, b, c >,\ < a, b, c, b, b >, \\ < b, a, b, c, b >,\ < b, b, a, b, c >, \\ < b, a, c, b, b >,\ < b, b, a, c, b > \end{array}$$

The orbit of label a is $\{1, 2, 3\}$, which partially overlaps with the orbit of label c, $\{3, 4, 5\}$. Consider an abstraction, ϕ, which maps a and c to the same abstract label:

$$\phi(a) = \phi(c) = a$$
$$\phi(b) = b$$

$\phi(< b, a, c, b, b >) =< b, a, a, b, b >$ and applying operator $\phi(o_2) = o_2$ to this state results in $< a, a, b, b, b >$, which has no pre-image in S_2.

The general lesson illustrated these examples is that labels with non-overlapping or partially overlapping orbits should not be mapped to the same abstract label.

[2] gives efficient algorithms for calculating orbits in permutation groups. When the state space is not a permutation group, the orbits may not be efficiently computable. An alternative to automatic computation of orbits is to determine them by hand and restrict the domain abstractions considered by the search procedure in Figure 1 so that it only maps labels with identical orbits to the same abstract label.

5.5 Blocks

Another structural property which interacts with domain abstraction and can cause non-surjectivity is a block structure of the labels. Informally, a block is a set of labels which always move in a co-ordinated manner.

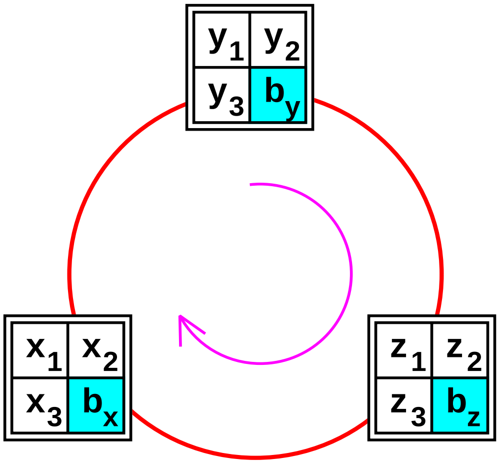


Fig. 16. Three 2×2 Sliding Tile Puzzles on a Ring

Consider the puzzle shown in Figure 16. There are three 2×2 puzzles arranged on a ring. The tiles can move as usual within each individual puzzles and the puzzles can be rotated on the ring, but there is no way for a tile in one puzzle to be exchanged with a tile in another puzzle. The set of labels representing each individual 2×2 puzzle forms a block.

To illustrate how block structure can interact with domain abstraction to produce non-surjectivity consider the search space S_3 defined as

$$seed = < a, b, c, d > \quad domain = \{a, b, c, d\}$$

$$succ = \left\{ \begin{array}{c} o_1 :< A, B, \_, \_ > \rightarrow < B, A, \_, \_ > \\ o_2 :< A, B, C, D > \rightarrow < C, D, A, B > \\ o_3 :< a, \_, \_, B > \rightarrow < a, a, B, B > \end{array} \right\}$$

S_3 has 12 states:

$$\left\{ \begin{array}{l} < a, b, c, d >, \; < c, d, a, b >, \; < b, a, c, d > \\ < a, a, d, d >, \; < c, d, b, a >, \; < d, c, a, b > \\ < d, d, a, a >, \; < a, b, d, c >, \; < d, c, b, a > \\ < a, a, c, c >, \; < b, a, d, c >, \; < c, c, a, a > \end{array} \right\}$$

Because of operators o_1 and o_2 all labels can reach all positions: there is no interesting orbit structure. But there is a block structure. $\{a, b\}$ is a block, as is $\{c, d\}$. Note that operator o_3 cannot be applied if the $\{c, d\}$ block is in the first two positions.

Now consider the domain abstraction ϕ which maps a and d to the same abstract label

$$\phi(a) = \phi(d) = a$$
$$\phi(b) = b$$
$$\phi(c) = c$$

$\phi(< d, c, a, b >) =< a, c, a, b >$ for which operator $\phi(o_3) = o_3$ applies and produces $< a, a, b, b >$ which has no pre-image in S.

Our notion of blocks is adapted from the blocks defined for Permutation Groups [5], which play an important role in reduced representations. As with orbits, for permutation groups there are efficient algorithms for calculating blocks [2]. When the search space is not a permutation group, the blocks may not be efficiently computable but, as in Figure 16, they are often evident to a human in the physical system that generates the space.

6 Representative Applications

This section presents three applications illustrating the use of domain abstraction to define heuristics for novel search spaces. The first two are commercially available combinatorial puzzles – the Skewb Cube has 6,298,560 states and the Pyraminx has 75,582,720 states[5] . In the third application, heuristics are created for the series of search spaces defined by macro-operator subgoals.

In all cases the orbits and blocks were identified easily by a hand inspection of the physical puzzle that gave rise to the search space and used to constrain the abstractions considered in order to avoid non-surjective abstractions.

The first two applications are essentially a manual execution of the procedure in Figure 1. The granularity of the abstractions was chosen by hand, by trying different granularities and seeing what size of pattern database they produced, and the abstractions of the chosen granularity were ranked by manually applying equation 6 (Section 3) to them.

The third application used an automatic procedure to search for abstractions that fit in memory.

[5] Skewb Cube and Pyraminx are copyrighted products of Uwe Meffert

6.1 Skewb Cube

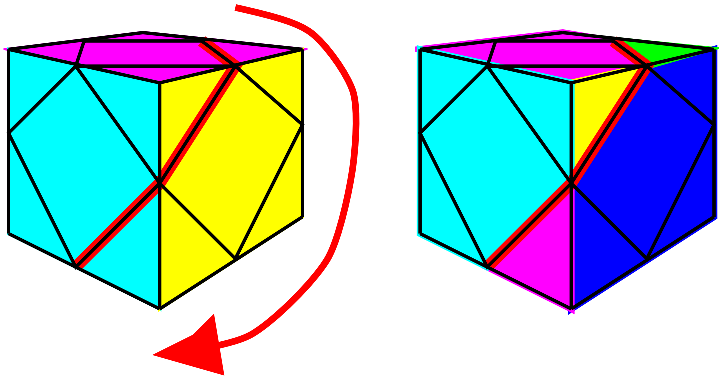


Fig. 17. Skewb Cube

The Skewb Cube has six colored faces. Each face has a diamond shaped middle piece and four triangular corner pieces. The operators twist half of the puzzle about the diagonals by 120° degrees.

The orbit and block properties are physically manifested in this puzzle. There is no way that a corner piece can be exchanged with a middle piece hence there must be at least two orbits. Because of the 120° degree rotations, the corner pieces move on two separate orbits. Thus there are three orbits altogether. If the labels in a state actually represent the colored stickers on the faces, it is also clear that every edge piece has three stickers glued on it, and hence those labels form a block. The block's orientation is uniquely determined by any one of the three stickers.

Our state representation for this puzzle consists of the thirty labels [6]. As discussed in section 5.4, we will create domain abstractions which avoid mapping labels on different orbits to the same abstract label, and because the corner blocks have an orientation, when we assign an abstract label to one of the stickers, the rest of the stickers on the same block will be assigned the same abstract label. In particular, we choose one of the orbits of the corners and paint all labels on three of the four corner pieces to the same new color. We also choose two of the four corner pieces from the other orbit and paint the labels on these to the same new color. Finally, we choose four of the five middle labels and assigned them a new color. This is a $GRAN^{\phi} =< 9, 6, 4 >$ abstraction. All such domain abstractions produce abstract spaces with 6480 states.

While there are a large number of such domain abstractions, they fall into 15 equivalence classes by isomorphism. We chose one representative from each of these 15 classes, and ranked them using equation 6. The best, the worst, and a medium ranking abstraction were each used to solve the same 100 easy (length

[6] one of the labels represents a middle piece that never moves

7), 100 medium (length 8) and 100 hard (length 9) start states. The average performance is tabulated in Table 4. Clearly these small pattern databases define very good heuristics for this space.

	Problems		
rank	**easy**	**medium**	**hard**
best	157	860	3383
medium	200	1138	6717
worst	272	1520	7668

Table 4
The Average of the Number of Nodes Expanded by the Best, a Medium and the Worst Ranked Abstractions on the Same 300 Start States.

6.2 Pyraminx

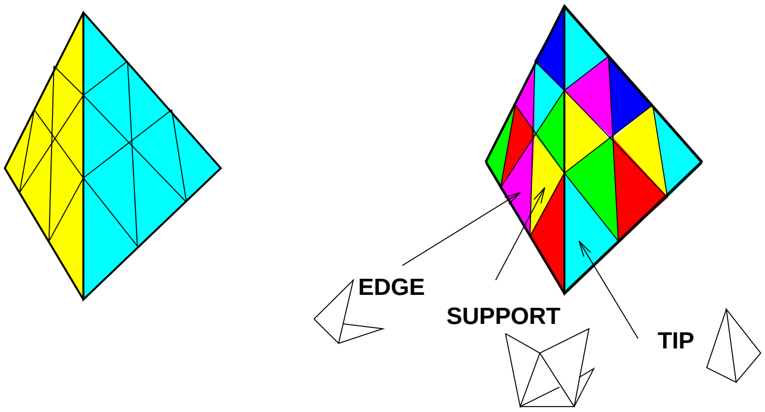


Fig. 18. Pyraminx

The Pyraminx is a Rubik's Cube style puzzle, but the pieces make up a pyramid. The number of different states is 75,582,720. There are three types of building blocks. The four *tip* pieces are themselves little solid pyramids which may be turned by 120° degrees. The tip pieces are mounted on the four *support* pieces which also have three colored stickers. In between support pieces are the six *edge* pieces which have two colored stickers. The pyramid also turns about each support piece by 120° degrees as well. The support pieces do not move with respect to each other but their orientation may be different. Because the tip pieces are only attached to the support pieces, they do not move with respect to each other but each can be oriented in three different ways. The middle pieces can be exchanged with each other and can be oriented in two different ways. The orbits and blocks – as with the Skewb Cube – physically manifest themselves. Each tip, support and edge piece is a block. Each of the tip and support pieces forms its own orbit because they do not move with respect to each other. The middle pieces altogether also form an orbit. It is also the case that the shortest path between two states can be decomposed into five shortest path problems: first fix the support and middle pieces and then fix the four tip pieces one by one. We do not take advantage of this fact even though it would significantly reduce search effort. Taking into account the orbits and blocks, we generated all abstractions which assign the same labels to every face of three of the four support pieces. The stickers on two of the four tip pieces also get a new label assigned and every sticker on three of six edge pieces also gets mapped to a new label.

The abstract spaces under isomorphism fall into 6 different equivalence classes. Each abstract space has 25920 states. We determined the A* branching factor to be 9.03. Using this we ranked a representative pattern database from each equivalence class using equation 6. We ran the best, a medium and the worst ranked heuristics on 50 easy (length 7), 50 medium (length 9) and 50 hard (length 11) problems. The results are tabulated in Table 5.

	Problems		
rank	**easy**	**medium**	**hard**
best	711	4535	38572
medium	1079	5718	42638
worst	1185	5728	42949

Table 5
The Average of the Number of Nodes Expanded by the Best, a Medium and the Worst Ranked Abstractions on the same 150 start states.

The difference between the best and worst heuristics is not as pronounced as with the Skewb Cube or the 8-Puzzle. Interestingly, equation 6 predicted this marginal ratio.

6.3 Optimal-length Macro-operators

This application is described fully in [11]. Here we just sketch the application and how automatically created heuristics were instrumental in its success.

Korf[18] introduced the "macro-operator" approach to very quickly constructing suboptimal solutions for search problems. A sequence of subgoals is defined: each subgoal requires fixing additional state variables to their final values while maintaining the values of the state variables fixed by previous subgoals.

Once the subgoal sequence is defined, the challenge is to find the macro-operators (sequence of operators) needed to achieve each subgoal. Since these macro-operators are going to be concatenated to create the final solution it is desirable to find the shortest possible macro-operators for the subgoals. However, when the original space is very large (e.g. Rubik's Cube in this application), the search spaces associated with many of the subgoals are also very large, and good, admissible heuristics for each of these spaces are needed in order to find optimal macro-operators.

An automatic method very similar to Figure 1 was used successfully to create heuristics to search for optimal-length macro-operators for Rubik's Cube [11]. The minimum abstraction was successively made more and more abstract until it created a pattern database smaller than the maximum allowed size. This automatic method was so successful that we were able to reduce the number of subgoals from 18 to 6 by merging successive subgoals. This led to a very significant reduction in the lengths of overall solutions, from an average of about 90 moves (5 times optimal) to an average of about 54 moves (3 times optimal).

This is an interesting example of a situation where it is highly desirable to create heuristics automatically. The spaces defined by the subgoals, and the corresponding heuristics, are of no interest once the macro-operators have been found. It is very convenient to have a fast automatic method for finding heuristics, rather than having to hand-craft heuristics for each of the subgoals.

7 Discussion

The methods and results of previous sections raise numerous general issues that will now be discussed. First, the positive contributions of our work will be summarized. Then its limitations and some possible extensions will be discussed.

7.1 Contributions

The heuristics created by domain abstraction have been seen to perform extremely well. In section 6 and in the original pattern database studies ([4], [19]), the pattern databases were the difference between feasibility and infeasibility. On the 8-Puzzle an average pattern database of size 3024 outperforms the venerable Manhattan distance heuristic.

Previous authors [16][23] have observed that their methods of abstraction, in some circumstances, are not able to create useful abstractions because the type of abstractions being considered are either too fine-grained, and therefore very expensive to compute, or too coarse-grained, and therefore of little use in guiding search. Although this depends to a large extent on how the search space is encoded, we believe domain abstraction suffers less from this problem than previous methods of abstraction. Typically it offers a fair range of granularities that are fine-grained enough to be useful but coarse enough that the corresponding pattern database fits in memory. Precisely controlling the size of the pattern database is difficult, in general. However, in certain cases, including all the surjective abstractions in this paper, the size can be exactly calculated, or at least estimated with reasonable accuracy, from a domain abstraction's granularity. This calculation requires knowledge of a space's orbits and blocks, which in general cannot be efficiently determined automatically. However, as was the case with the Skewb Cube and the Pyraminx puzzles, the orbits and blocks are often evident from the physical structure of the system that gives rise to the search space.

As illustrated in Section 6, domain abstraction, the formula for ranking pattern databases (equation 6), and the procedure in Figure 1 are useful for creating heuristics for novel spaces even if applied manually.

7.2 Limitations and Extensions

We believe that the methods presented in this paper will lead to a successful system that uses domain abstraction to create heuristics automatically. The one application we have attempted was highly successful [11], but this is too little experience to draw general conclusions. We anticipate an automatic method will be most successful when applied to novel search spaces, for which no good heuristic is known. In such cases, we expect the computational cost of finding a pattern database that fits in memory to be less than the cost of solving one problem in the original space without a heuristic.

In [11] the automated heuristic-creation process stopped as soon as it found the first pattern database that would fir in memory. It is an open question whether it is worthwhile to search extensively for higher scoring pattern databases after this point. For the spaces studied in this paper the best heuristic of a given size is only about twice as good as the worst, implying there is not a strong motivation for evaluating a large number of abstractions. We did not do any such search

The greatest obstacle one encounters in using domain abstraction is that non-surjective abstractions arise often if steps are not taken to avoid them. It was shown that mapping labels on different orbits, or in different blocks, to the same abstract label is never helpful and might result in a non-surjective abstraction. Unfortunately we do not at this time have a complete characterization of the causes of non-surjectivity. Although it is tempting to regard non-surjectivity as a flaw in the domain abstraction, one example was presented – the dual encoding of the 8-Puzzle – in which non-surjectivity was seemingly inherent in the way the problem was encoded.

By its very nature domain abstraction is most useful in domains with many labels. For example if states are represented by binary vectors domain abstraction is entirely useless. To overcome this limitation, we are beginning to investigate other methods of abstraction.

There are other techniques for using memory to speed up heuristic search. [15] provides a good summary and an initial comparison of some of the techniques. These techniques can be used in combination with each other and with pattern databases. The optimal allocation of memory among these techniques is an open research question.

In our experiments a single pattern database was used to guide search. This was done in order to eliminate confounding factors in interpreting the results. In practice, pattern databases would be used in conjunction with other knowledge of the search space. For example, [4] uses hand-crafted pattern databases in combination with the Manhattan distance and exploits symmetries in the search space and the invertibility of the operators to decrease the size and increase the usefulness of the pattern databases. [11] and [19] use multiple pattern databases simultaneously.

8 Conclusion

Previous work has shown the pattern database approach[4] to be a highly successful method for hand-crafting good heuristics. In this paper we have shown that this is a very promising approach for creating heuristics automatically. The paper describes domain abstraction, which extends the notion of "pattern" in previous pattern database research in a simple way which permits available memory to be much more fully exploited to reduce search time.

The paper has presented two of the key elements required to effectively search through the space space of possible pattern databases to find ones that both fit in memory and are good heuristics. The first is a mechanism for increasing or decreasing the "abstractness" of an abstraction. The second is a formula for scoring pattern databases such that higher scores tend to indicate better heuristic performance.

The scoring function presented is based on a formula developed by Korf and Reid [20]. Our contribution was to demonstrate that a particular approximation to this formula, easily computed from a given pattern database, is monotonically related to the actual number of nodes expanded using the pattern database. This demonstration took the form of a large-scale experiment involving all possible domain abstractions for the 8-Puzzle in which the blank tile remains unique.

An important contribution of the paper was identifying the remaining obstacles to automatic heuristic creation. Foremost among these is the problem of non-surjectivity: domain abstractions can create abstract spaces in which some states do not have a pre-image. We have identified two causes of non-surjectivity, related to the space's orbits and blocks, but we also showed that there are other causes which are not presently well understood. Several other limitations and possible extensions have been discussed.

9 Acknowledgments

Thanks to Jonathan Schaeffer and Joe Culberson for their encouragement and helpful comments and to Richard Korf for communicating his unpublished extensions of the [20] work.

References

[1] R. B. Banerji. *Artificial Intelligence: A Theoretical Approach.* North Holland, 1980.

[2] G. Butler. *Fundamental Algorithms for Permutation Groups.* Lecture Notes in Computer Science. Springer-Verlag, 1991.

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

[4] J. C. Culberson and J. Schaeffer. Searching with pattern databases. *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)*, pages 402–416, 1996.

[5] J. D. Dixon and B. Mortimer. *Permutation Groups.* Graduate Texts in Mathematics. Springer-Verlag, 1996.

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

[7] S. Edelkamp and R. E. Korf. The branching factor of regular spaces. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 299–304, 1998.

[8] J. Gasching. A problem similarity approach to devising heuristics: First results. *IJCAI*, pages 301–307, 1979.

[9] G. Guida and M. Somalvico. A method for computing heuristics in problem solving. *Information Sciences*, 19:251–259, 1979.

[10] 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.

[11] I. T. Hernádvölgyi. 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)*, pages 194–203, 2001.

[12] 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.

[13] R. C. Holte and I. T. Hernádvölgyi. A space-time tradeoff for memory-based heuristics. *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 704–709, 1999.

[14] 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.

[15] H. Kaindl, G. Kainz, A. Leeb, and H. Smetana. How to use limited memory in heuristic search. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 236–242, 1995.

[16] D. Kibler. Natural generation of admissible heuristics. Technical Report TR-188, University of California at Irvine, July 1982.

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

[18] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[19] 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.

[20] R. E. Korf and M. Reid. Complexity analysis of admissible heuristic search. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 305–310, 1998.

[21] J. Mostow and A. Prieditis. Discovering admissible heuristics by abstracting and optimizing: A transformational approach. *IJCAI*, pages 701–707, 1989.

[22] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison & Wesley, 1984.

[23] A. E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.

[24] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, pages 47–59, 1984.