Hierarchical A*: Searching Abstraction Hierarchies Efficiently

R.C. Holte, M.B. Perez, R.M. Zimmer, A.J. MacDonald

Abstract

Abstraction, in search, problem solving, and planning, works by replacing one state space by another (the "abstract" space) that is easier to search. The results of the search in the abstract space are used to guide search in the original space. A natural application of this technique is to use the length of the abstract solution as a heuristic for A in searching in the original space. However, there are two obstacles to making this work efficiently. The first is a theorem [Valtorta, 1984] stating that for a large class of abstractions, "embedding abstractions," every state expanded by blind search must also be expanded by A* when its heuristic is computed in this way. The second obstacle arises because in solving a problem A* needs repeatedly to do a full search of the abstract space while computing its heuristic. This paper introduces a new abstraction-induced search technique, "Hierarchical A*," that gets around both of these difficulties: first, by drawing from a different class of abstractions, "homomorphism abstractions," and, secondly, by using novel caching techniques to avoid repeatedly expanding the same states in successive searches in the abstract space. Hierarchical A* outperforms blind search on all the search spaces studied.*

Hierarchical A*: Searching Abstraction Hierarchies Efficiently

R.C. Holte[1], M.B. Perez[1], R.M. Zimmer[2], A.J. MacDonald[3]

1. Introduction

Several researchers have investigated abstraction as a means of automatically creating admissible heuristics for A* search [Gaschnig, 1979; Pearl, 1984; Prieditis, 1993; Guida and Somalvico, 1979]. The general idea is to equate h(S), the estimate of the distance in the state space SS from state S to the goal G, with the exact distance in some other state space, SS′ — the "abstraction" of SS — from the state in SS′ corresponding to S to the state corresponding to G. As long as the distance between every pair of states in SS is greater than or equal to the distance between the corresponding states in SS′ h(S), defined in this way, is an admissible heuristic. [Prieditis,1993] uses this observation to define an "abstraction transformation": ϕ is an abstraction transformation if the distance between any two states, S_1 and S_2, in SS is guaranteed to be greater than or equal the distance between $\phi(S_1)$ and $\phi(S_2)$ in $\phi(SS)$[4].

The earliest and most commonly studied type of abstraction transformation is the "embedding". Informally, ϕ is an embedding transformation if it "adds edges" to SS. For example, techniques that add macro-operators to, or drop preconditions from a state space definition are embeddings. The other main type of abstraction transformation is the "homomorphism". Informally, a homomorphism ϕ groups together several states in SS to create a single abstract state. For example, techniques that drop a predicate entirely from a state space description [Knoblock,1994] are homomorphisms.

The aim of creating a heuristic is to speed up search. Without a heuristic, A* blindly searches in the original space. With a heuristic, A*'s search will be more focused, and the search effort in the original space will be reduced by some amount (the "saving"). The primary risk in using a heuristic created by abstraction is that the cost of computing h(-) can exceed the saving. If this happens the use of the heuristic is harmful: the cost to compute the heuristic outweighs the benefit derived from using it.

[1] Computer Science Dept., University of Ottawa, Ottawa, Ontario, Canada K1N 6N5. holte@csi.uottawa.ca

[2] Computer Science Dept., Brunel University, Uxbridge, England UB8 3PH. Robert.Zimmer@brunel.ac.uk

[3] Electrical Engg. Dept., Brunel University, Uxbridge, England UB8 3PH. Alan.MacDonald@brunel.ac.uk

[4] the definition in [Prieditis,1993] only requires this property to hold when S_2 is a goal state. We use the more strict definition because we do not assume, as [Prieditis,1993] does, that the goal is known at the time abstractions are being created.

A cost-benefit analysis of heuristics automatically generated by embedding transformations was presented in [Valtorta,1984]. It was proven that if SS is embedded in SS′ and h(-) is computed by blindly searching in SS′, then A* using h(-) will expand every state that is expanded by blindly searching directly in SS. In other words, this method of defining a heuristic cannot possibly speed up search – the total number of nodes[5] expanded using the heuristic (including those expanded in the abstract space) must equal or exceed the number expanded when blindly searching in the original space. Based on this theorem, we define "Valtorta's Barrier" to be the number of nodes expanded when blindly searching in a space. Valtorta's theorem states that this barrier cannot be "broken" using any embedding transformation.

To date the only system that has succeeded in breaking Valtorta's barrier is Absolver II [Prieditis,1993]. Given the state spaces associated with the Fool's Disk and Instant Insanity puzzles and the particular goals to be achieved, Absolver II created cost-effective heuristics using abstraction transformations alone. On the other 11 problems to which it was applied, abstraction by itself was unable to create a cost-effective heuristic (on 5 of these cost-effective heuristics were created when abstraction was used in conjunction with "speed up" transformations).

In this paper we show that Valtorta's barrier can be broken on a wide variety of search spaces using a general-purpose homomorphic abstraction technique. Section 2 presents a generalized version of Valtorta's theorem that applies to all types of abstraction transformation. From this theorem it is clear that is possible, at least in principle, for homomorphic abstractions to break Valtorta's barrier. Section 3 then introduces our particular homomorphic abstraction technique and initial experimental results. The results are strongly negative: search using the abstraction hierarchy expands many **more** states than blind search – over 17 times as many in one of the testbed search spaces. The cause of these negative results is identified and two complementary approaches to alleviating the problem are investigated. Section 4 presents an algorithmic approach. The A* algorithm is customized for searching in abstraction hierarchies. The customized version, called Hierarchical A*, expands roughly 6 times fewer states than the original A* and breaks Valtorta's barrier in about half the testbed search spaces. Section 5 presents the second approach, which is to create abstractions that are less fine-grained. This also breaks Valtorta's barrier for some of the testbed search spaces. When the two approaches are combined, Valtorta's barrier is broken for all the testbed search spaces.

2. Valtorta's Theorem Generalized

The main theorem in [Valtorta,1984], which is specific to embeddings, is easily generalized to any abstraction transformation. The generalized theorem is this:

[5] the terms "node" and "state" are synonymous. We use "state" everywhere except in referring to the number of states expanded, where we use the standard term "nodes expanded".

Let S be any state necessarily expanded[6] when the given problem (Start,Goal) is solved by blind search directly in state space SS,
let ϕ be any abstraction mapping from SS to SS′ and
let $h_\phi(S)$ be computed by blindly searching in SS′ from $\phi(S)$ to $\phi(Goal)$.
If the problem is solved in SS by A* search using $h_\phi(-)$, then either:

(1) S itself will be expanded, or
(2) $\phi(S)$ will be expanded.

Proof: see appendix A.

When ϕ is an embedding, $\phi(S)=S$ and we get Valtorta's theorem: every state necessarily expanded by blind search is also necessarily expanded by A* using $h_\phi(-)$. However, if ϕ is a homomorphism speedup is possible because the expansion of many states in the original space can be replaced by the expansion of a single state in the abstract space.

A simple example illustrates that heuristics based on homomorphic abstractions can indeed greatly reduce the number of nodes expanded. Suppose state space SS is an N×N grid and that the problem being solved is to get from the bottom left corner (1,1) to the bottom right corner (N,1). To solve this problem with blind search, it is necessary to expand all states at distance (N-1) or less; thus $O(N^2)$ states are expanded. Let ϕ be the mapping that ignores the second co−ordinate. $\phi(SS)$, then, is a linear space with states (1), (2),... (N). Computing $h_\phi(Start)$ requires finding a path in SS′ from (1) to (N). $O(N)$ states are expanded during this search, but notice that this search generates the value of $h_\phi(S)$ for all S, so no further search need be done in the abstract space. Furthermore, the $h_\phi(-)$ defined by this particular abstraction is a perfect heuristic, and therefore the search in SS will expand only those states on the solution path. Thus, the total number of nodes expanded by A* using $h_\phi(-)$ is $O(N)$, far fewer than the $O(N^2)$ nodes expanded by blind search.

The following generalization of a theorem in [Valtorta,1984] is also useful:

Let ϕ be any abstraction mapping from SS to SS′ and
let $h_\phi(S)$ be computed by blindly searching in SS′ from $\phi(S)$ to $\phi(Goal)$.

Then $h_\phi(S)$ is a monotone (consistent) heuristic.

Proof: see appendix A.

[6] a state is necessarily expanded by blind search if its distance from Start is strictly less than the distance from Start to Goal.

3. Hierarchical Search using A*

Abstractions are created in the current system using the "max-degree" STAR abstraction technique described in [Holte et al.,1996]. This technique is very simple: the state with the largest degree is grouped together with its neighbours within a certain distance (the "abstraction radius") to form a single abstract state. This is repeated until all states have been assigned to some abstract state. Having thus created one level of abstraction the process is repeated recursively until a level is created containing just one state. This forms an abstraction hierarchy whose top-level is the trivial search space. The bottom, or "base", level of the hierarchy is the original search space. This method of creating abstraction was originally designed and has proven very successful for the search technique known as "refinement" (see [Holte et al.,1996], [Holte et al.,1994]). It was not known at the outset of the present work whether it would or would not create abstraction hierarchies suitable for A* search. It certainly is not suitable for abstracting search spaces in which different operators have different costs, but that consideration does not arise in the experiments in this paper.

The implementation of A* is standard except that in estimating the distance to the goal from a non-goal state, S, it uses the minimum cost of the operators applicable to S in addition to whatever other heuristic estimates might be available. When multiple sources of heuristic information are available, they are combined by taking their maximum. This way of combining multiple heuristics is guaranteed to be admissible if the individual heuristics are admissible, but it may not be monotone even though the individual heuristics are (for an example to the contrary see the next section). However, it is easy to see that combining the minimum operator cost with any monotone heuristic produces a monotone heuristic if the operator costs are symmetric (i.e. every operator has an inverse of the same cost), as is the case in the search spaces in our experiments.

The "cheapest operator" information is most useful when no other heuristic information is available (i.e. Blind Search, A* with h(S)=0 for all S). In this case its use considerably reduces the number of nodes expanded. When other sources of heuristic information are available the "cheapest operator" information is only useful for states estimated by the other means to be very close to the goal.

Hierarchical search using A* is straightforward. As usual, at each step of the A* search a state is removed from the OPEN list and "expanded", i.e. each of its successors is added to the OPEN list (if it has not previously been opened). In order to add a state S to the OPEN list, h(S) must be known. This is is computed by searching at the next higher level of abstraction, using the abstract state corresponding to S (call this state ϕ(S)) as the abstract start state and the abstract state corresponding to the goal, ϕ(goal), as the abstract goal. When an abstract solution path is found, the exact abstract distance from ϕ(S) to ϕ(goal) is known. As described in the preceding paragraphs this is combined with other estimates (e.g. the cost of the cheapest operator applicable to S) to produce the final h(S) value.

Note that when the abstract path from ϕ(S) to ϕ(goal) is found, exact abstract distance-to-goal information is known for **all** abstract states on this path. And each of these abstracts states corresponds, in general, to many states in the level "below" (the state space containing S). Therefore, a single abstract search produces heuristic estimates for many states. All this information is cached. If an h(-) value is needed for any of these states, the value is simply looked up without any search being done at the abstract level. This idea was illustrated in the example in section 2, where a single search at the abstract level produced h(-) information for all the base level states. Despite this caching technique, it is generally true that to solve a single base level problem, a hierarchical search technique will need to solve many problems at the first level of abstraction. And each one of these abstract problems will require solving many problems at the next level of abstraction, etc. The number of nodes expanded to solve a single base level problem is the total number of states expanded during all searches related to that base level problem at all levels of abstraction (including the base level itself).

The hierarchical search technique just described will be referred to as naive hierarchical A*, to contrast it with the versions presented in the next section. The various hierarchical A* techniques were evaluated empirically on 8 state spaces (described in appendix C). All have invertible operators and the cost of applying all operators is the same (1). Test problems for each state space were generated by choosing 100 pairs of states at random. Each pair of states, <S1,S2>, defined two problems to be solved: <start=S1,goal=S2> and <start=S2,goal=S1>. To permit detailed comparison the same 200 problems were used in every experiment. The "nodes expanded" results shown are averages over these 200 problems.

Naive hierarchical A* expands many more states than Blind Search in every testbed search space (see Table 1). The clue to understanding why this happens is the fact that naive hierarchical A* expands many more states than there are states in the entire abstraction hierarchy. This implies that the some states are being expanded many times in the course of solving a single base level problem. Since A* with a monotone heuristic never expands the same state twice in a single search, we may conclude that the duplication arises from the same states being expanded on many different searches associated with the same base level search. The next section presents methods for greatly reducing this sort of duplication.

The number of nodes expanded in the base level by hierarchical A* is a fundamental limit on the amount of speedup that A* can achieve with the given abstraction hierarchy. This number indicates the quality of the heuristic, i.e. how much the heuristic reduces the search effort in the original space. For example, in the Permute-6 search space hierarchical A* expanded 77 states in the base level, a reduction of 75% compared to the number expanded by Blind Search. This is a modest savings. And yet this is the best of the heuristics in Table 1. At the other extreme is the heuristic created for the Missionaries and Cannibals search space (MC 60-40-7) which produced a saving of only 25%. The quality of the heuristics is a property of the abstraction technique and can only be improved by using a different abstraction technique. Except for varying the granularity of abstraction (section 5), this topic is beyond the scope of the present paper but is clearly a key area for future research.

TABLE 1. **Naive Hierarchical A*.** (abstraction radius = 2)					
Search Space	Size (# states)		Nodes Expanded		
	All Levels	Base Level	Blind Search	Hierarchical A* All Levels	Hierarchical A* Base Level
Blocks-5	1166	866	389	2766	118
5-puzzle	961	720	348	3119	224
Fool's Disk	4709	4096	1635	12680	629
Hanoi-7	2894	2187	1069	18829	701
KL2000	3107	2736	1236	7059	641
MC 60-40-7	2023	1878	934	2412	702
Permute-6	731	720	286	806	77
Words	5330	4493	1923	19386	604

The fact that the heuristics produced by the current abstraction technique result in only a modest reduction in the number of nodes expanded at the base level makes breaking Valtorta's barrier particularly challenging. In order to break the barrier the total number of nodes expanded in all the abstract searches must be less than the savings the heuristics produce. For example, consider the 5-puzzle. The heuristic created by abstraction results in A* expanding 224 states at the base level, a saving of 124 states over Blind Search. In order to break Valtorta's barrier the cost (in terms of nodes expanded at all the abstract levels) of computing the heuristic for all these 224 states (and the additional states that were opened but never closed) must not exceed 124. This is not impossible: expanding a few states at the abstract level can provide heuristic values for many states at the base level, as the example in section 2 illustrates.

4. Customizing A* for Hierarchical Search

In hierarchical search, a single base level search can spawn a large number of searches at the abstract levels. As the preceding results show, these searches will often expand many of the same states. The key to reducing this duplication is the observation that **all searches related to the same base level search have the same goal**. The most obvious way to exploit this fact is to cache whatever h(-) values are computed and to use them, without recomputing them, until the goal changes. This simple caching strategy is implemented in naive hierarchical A*.

When a search at an abstract level terminates the exact distance to the goal, h*(S), is known for every state on the solution path. This information can be cached and used in lieu of h(S) in subsequent searches with the same goal. This improves the quality of the heuristic and therefore may reduce the number of nodes expanded by subsequent searches. This technique is called h*-caching. As can be seen in Table 2 (column V1) this roughly halves the number of nodes expanded.

TABLE 2. **Hierarchical A*.** (abstraction radius = 2)						
	Nodes Expanded					# problems
Search	Blind	Hierarchical A*				V3 < BS
Space	Search	Naive	V1	V2	V3	(out of 200)
Blocks-5	389	2766	1235	478	402	96
5-puzzle	348	3119	1616	854	560	14
Fool's Disk	1635	12680	8612	3950	**1525**	132
Hanoi-7	1069	18829	10667	5357	3174	0
KL2000	1236	7059	3490	1596	**1028**	171
MC 60-40-7	934	2412	1531	1154	**863**	128
Permute-6	286	806	482	**279**	**242**	113
Words	1923	19386	7591	2849	**1410**	124

Note that the heuristic produced by h*-caching is not monotone. For example, suppose all operators have cost 1 and that h(S)=1 for all non-goal states. After solving one problem and caching the h*(-) values, states on the solution path may have quite large h(-) values, while their neighbours off the solution path will still have h(-)=1. Thus the difference between neighbouring h(-) values will be greater than 1, the "distance" between the two states. Therefore, the new h(-) function will not satisfy the definition of monotone.

Because the h(-) function is not monotone it is possible for a state to be closed prematurely, i.e. closed before the shortest path to it from the start state has been found. With non-monotone heuristics it is, in general, necessary to re-open such states in order to be guaranteed of finding the shortest path. However, with the type of non-monotone heuristic produced by h*-caching this is not necessary and in fact our A* implementation has no provision for re-opening closed states. To see this let S be the start state and P a prematurely closed state. For P to be prematurely closed it must be that h*(P) is not known and that each shortest path from S to P passes through some state, X, for which h*(-) is known. If no such X is on any shortest path from S to Goal then neither is P, so P's being prematurely closed is irrelevant. On the other hand, suppose the shortest path from S to Goal passes through such an X. The fact that h*(X) is known means that in a previous search X was on the solution path and therefore a shortest path from X to Goal was previously discovered. For every state on this path h*(-) is known. Therefore none of these states has yet been expanded in the current search and so this segment of the shortest path from S to Goal will be found without having to re-open any closed states.

As just noted, for every state X for which h*(X) is known a shortest path from X to Goal is also known. If this path is cached in addition to h*(X) there is no need to expand X. This is because knowing a path of length g(X) to X is equivalent to knowing a path of length g(X)+h*(X) to Goal. Thus, instead of adding X to the OPEN list, one can add Goal instead and terminate search, as usual, when Goal is atop the OPEN list. This technique is called optimal-path caching. Column V2 in Table 2 shows the number of nodes expanded when optimal-path caching is added to naive hierarchical A*. Optimal-path caching produces about double the savings of h*-caching, and breaks Valtorta's barrier in the Permute-6 search space.

The final technique pertains to states that were opened (or closed) during search but are not on the solution path. For each such state, S, a distance from the start, g(S), is known. If the solution path is length P, then by its optimality it must be that $P \leq g(S) + h^*(S)$. Re-arranging this as $P-g(S) \leq h^*(S)$ we see that $P-g(S)$ is an admissible heuristic, which we call the P-g heuristic. When all operators have inverses it can be shown (see Appendix B) that combining the P-g heuristic with any monotone heuristic produces a monotone heuristic. Note that P-g caching subsumes h*-caching because $P-g(S) = h^*(S)$ when S is on the solution path. Also note that it is not necessary to compute $P-g(S)$ if S is open when the search terminates because in this case $P \leq g(S)+h(S)$ and therefore $P-g(S) \leq h(S)$.

The system V3 includes optimal-path caching and P-g caching. It breaks Valtorta's barrier in 5 of the 8 search spaces used for testing (the bold figures in Table 2). This shows that with homomorphic abstractions it is possible to achieve in practice what is theoretically impossible to achieve with embeddings. The rightmost column indicates the number of problems on which V3 broke Valtorta's barrier. The barrier is broken on at least some problems in all spaces except the Towers of Hanoi (Hanoi-7).

5. Varying the Granularity of Abstraction

Our abstraction technique allows us to control the granularity of the abstractions created by setting the abstraction radius. The preceding results were obtained with an abstraction radius of 2, which means a state is grouped with its immediate neighbours.

The net effect on "nodes expanded" of increasing the abstraction radius is not clear, because two antagonistic effects interact. On one hand, a larger radius means that the abstract spaces contain fewer states and also that a single abstract search produces heuristic values for more states. These factors will reduce the number of abstract nodes expanded by hierarchical A*. On the other hand, a larger radius means the heuristic is less discriminating: this tends to increase the number of nodes expanded (if the heuristic is completely undiscriminating, A* degenerates to Blind Search). [Prieditis and Davis, 1995] presents an initial quantitative analysis of the relation between "abstractness" (i.e. granularity) and the accuracy of the resulting heuristics.

Experiments were run with the abstraction radius set at values 2 through 5, and at larger values for the search spaces with large diameters. The number of nodes expanded by hierarchical A* decreased as the radius increased until a minimum was reached; increasing the radius beyond this point caused the number of nodes expanded to increase. Table 3 shows the results of the best radius found for V3 for each search space (the best radius for naive hierarchical A* was sometimes larger). V3 now breaks Valtorta's barrier on over half the problems in every search space (and on over 95% of the problems in 2 of the spaces).

TABLE 3. Hierarchical A*. (best abstraction radius)							
Search Space	Radius	Nodes Expanded			# problems	CPU seconds	
		Blind Search	Hierarchical A* Naive	Hierarchical A* V3	V3 < BS (out of 200)	Blind Search	V3
Blocks-5	5	389	611	**309**	123	69	86
5-puzzle	12	348	354	**340**	131	36	40
Fool's Disk	4	1635	**1318**	**1172**	194	872	902
Hanoi-7	20	1069	1097	**1055**	117	102	108
KL2000	5	1236	1306	**1072**	178	398	384
MC 60-40-7	4	934	**822**	**803**	144	266	253
Permute-6	5	286	**201**	**194**	192	82	67
Words	3	1923	9184	**1356**	128	1169	1273

Although the best radii are not, in most cases, large numbers, in every case they represent a significant fraction of the search space diameter. The abstraction hierarchies thus created had only one non-trivial abstract level, and it contained a small number of states. It is surprising that so coarse a heuristic is able to reduce the number of nodes expanded.

A consequence of there being just a single small non-trivial abstract level is that the algorithmic enhancements introduced in section 4 have a much reduced effect: in several of the spaces naive hierarchical A* expands only slightly more states than V3. However, the algorithmic enhancements are important because with most techniques for creating abstractions there is no easy way to control the granularity of the abstractions produced. Unlike naive hierarchical A*, V3 is robust: it performs well on abstractions of any granularity and is therefore the search algorithm of choice when granularity cannot be controlled.

"Nodes expanded" is a convenient theoretical measure, but it does not completely capture all the "work" that is done by a search system, especially by a hierarchical system that must repeatedly initialize the searches at the abstract levels and pass information from one level to the next. The actual speed of a search system depends on all these operations. Table 3 reports the CPU time taken by Blind Search and V3 to solve all 200 problems. Of course, one must be very cautious in interpreting CPU time results, as they can be heavily affected by low-level implementation details. In the present implementation, which is rather clumsy in many of its low-level details, it appears that the reduction in nodes expanded produced by V3 almost perfectly balances the additional overheads involved in hierarchical search.

6. Conclusions

In this paper we have shown that A* search using heuristics created automatically by homomorphic abstractions can "break Valtorta's barrier", that is, it can outperform blind search in terms of number of nodes expanded. This was accomplished in all the state spaces used in the experiments. To achieve this it was necessary to add two novel caching techniques to A* (optimal-path caching and P-g caching) and also, in some cases, to choose abstractions of suitable granularity. The reduction in number of nodes expanded was not large, but this was due to limitations of the abstraction technique and not to excessive algorithmic overheads. The development of an abstraction technique well-suited to hierarchical A* search is an important topic for future research.

Acknowledgements

This research was supported in part by an operating grant from the Natural Sciences and Engineering Research Council of Canada. The software was written in part by Denys Duchier and Chris Drummond. Thanks to Marco Valtorta for his encouragement, helpful comments, and the prototype of the example in section 2.

References

Gaschnig, J. (1979), "A Problem Similarity Approach to Devising Heuristics: First Results", *Proc. IJCAI'79*, pp. 301-307.

Holte, R.C., T. Mkadmi, R.M. Zimmer, and A.J. MacDonald (1996), "Speeding Up Problem-Solving by Abstraction: A Graph-Oriented Approach". to appear in the special issue of *Artificial Intelligence* on Empirical AI, edited by Paul Cohen and Bruce Porter.

Holte, R.C., C. Drummond, M.B. Perez, R.M. Zimmer, and A.J. MacDonald (1994), "Searching with Abstractions: A Unifying Framework and New High-Performance Algorithm", *Proc. of the 10th Canadian Conference on Artificial Intelligence (AI'94)*, Morgan Kaufman Publishers, pp. 263-270.

Knoblock, C.A. (1994), "Automatically Generating Abstractions for Planning", Artificial Intelligence, vol. 68(2), pp. 243-302.

Pearl, J. (1984), *Heuristics*, Addison-Wesley.

Prieditis, A. and R. Davis (1995), "Quantitatively relating abstractness to the accuracy of admissible heuristics", *Artificial Intelligence"*, vol. 74, pp. 165-175.

Prieditis, A. (1993), "Machine Discovery of Admissible Heuristics", *Machine Learning*, vol.12, pp. 117-142.

Guida, G. and M. Somalvico (1979), "A method for computing Heuristics in Problem Solving", *Information Sciences*, vol. 19, pp. 251-259.

Valtorta, M. (1984), "A result on the computational complexity of heuristic estimates for the A* algorithm", *Information Sciences*, vol. 34, pp. 48-59.

Appendix A. Generalization of Valtorta's Theorems

Definitions and Notation
ϕ = an abstraction mapping from search space **SS** to search space **SS′**.
$\mathbf{d(S_1,S_2)}$ = the length of the shortest path (in SS) from S_1 to S_2.
$\mathbf{d_\phi(S_1,S_2)}$ = the length of the shortest path (in SS′) from $\phi(S_1)$ to $\phi(S_2)$.
h*(S) = d(S,Goal).
$\mathbf{h_\phi(S)}$ = d_ϕ(S,Goal). h_ϕ(S) is computed by searching blindly in SS′ from ϕ(S) to ϕ(Goal).
state S is **necessarily expanded** by blind search iff d(Start,S) < h*(Start).
state S is **necessarily expanded** by A* iff d(Start,S) + h_ϕ(S) < h*(Start).

Main Theorem

Let S be any state necessarily expanded when the problem (Start,Goal) is solved by blind search directly in state space SS, and let ϕ be any abstraction mapping. If the problem is solved using A* search in SS then either S itself will be expanded or ϕ(S) will be expanded.

Proof:
When A* terminates, S will either be closed, open, or unvisited. If S is closed, it will have been expanded. If S is open, then h_ϕ(S) must have been computed during search (as part of adding S to the open list). h_ϕ(S) is computed by searching in SS′ starting at ϕ(S). If $\phi(S) \neq \phi(Goal)$ the first step in this search is to expand ϕ(S). On the other hand, if $\phi(S) = \phi(Goal)$ then $h_\phi(S) = 0$ and S itself is necessarily expanded. Finally, suppose S is unvisited (case 3 in [Valtorta,1984]'s proof). In this case on every path from Start to S there must be a state that was added to the open list during search but never expanded. Let M be any such state on any shortest path from Start to S. Because M was opened, h_ϕ(M) must have been computed. We will now show that in computing h_ϕ(M), ϕ(S) is necessarily expanded.

From the fact that S is necessarily expanded by blind search we have **d(Start,S) < h*(Start)**.
Because M is on a shortest path to S this can be rewritten **d(Start,M) + d(M,S) < h*(Start)**.
From the fact that M was never expanded we have $\mathbf{d(Start,M) + h_\phi(M) \geq h^*(Start)}$
Combining these last two inequalities we get: $\mathbf{d(M,S) < h_\phi(M) = d_\phi(M,Goal)}$.
Because ϕ is an abstraction mapping, we have $\mathbf{d_\phi(M,S) \leq d(M,S)}$.
Combining this with the previous inequality gives $\mathbf{d_\phi(M,S) < d_\phi(M,Goal)}$.
Thus ϕ(S) is necessarily expanded during the computation of h_ϕ(M).

In summary, whether S is closed, open, or unvisited at the termination of the A* search, either S itself or ϕ(S) will have been expanded during the search.

Consistency Theorem (the original is on p. 52, [Valtorta,1984])

h_ϕ(S), as defined above, is **consistent**, i.e. for all S and S′ ∈ SS: $h_\phi(S) \leq d(S,S') + h_\phi(S')$.

Proof:
Because d_ϕ(S,Goal) is the length of the shortest path: $\mathbf{d_\phi(S,Goal) \leq d_\phi(S,S') + d_\phi(S',Goal)}$ for all S and S′.
Substituting h_ϕ(-) for all occurrences of d_ϕ(-,Goal) gives: $\mathbf{h_\phi(S) \leq d_\phi(S,S') + h_\phi(S')}$ for all S and S′.
Because ϕ is an abstraction, $d_\phi(S,S') \leq d(S,S')$ and therefore $\mathbf{h_\phi(S) \leq d(S,S') + h_\phi(S')}$ for all S and S′.

Theorem 8 (p. 83) in [Pearl, 1984] proves that consistency and monotonicity are equivalent.

Appendix B. P-g Caching Preserves Monotonicity

Definitions and Notation

$d(S_1,S_2)$ = the length of the shortest path from S_1 to S_2.
g*(S) = d(Start,S)
g(S) = length of the shortest known path from Start to S at the time search terminates. This function is only defined if S is open or closed; if S is unvisited when search terminates, g(S) is undefined.
P = d(Start,Goal)
Pg(S) = P – g(S). Pg(S) is defined iff g(S) is defined.
h(S) = a monotone heuristic.
h′(S) = maximum { h(S), Pg(S) } if Pg is defined; otherwise h′(S) = h(S).

Lemma 1
If S is open when search terminates h′(S) = h(S).

Proof: S open when search terminates means **g(S) + h(S) ≥ P**. Thus **h(S) ≥ P – g(S) = Pg(S)**.

Lemma 2
If S is closed when search terminates h′(S) = Pg(S).

Proof: S closed when search terminates means **g(S) + h(S) ≤ P**. Thus **h(S) ≤ P – g(S) = Pg(S)**.

Theorem

Assuming that the operators defining the search space are invertible, if h(-) is monotonic then so is h′(-).

Proof:
Let S be any state and S′ any neighbour of S. We are required to show that h′(S) – h′(S′) ≤ d(S,S′).
If neither S nor S′ is closed then the result follows from the monotonicity of h(-) and the fact that h′(X) = h(X) unless X is closed. Suppose, then, that one (or both) of them is closed. Because of the assumption that operators are invertible we can, without loss of generality, assume that S is closed and that S′ is either closed or open (but not unvisited). Thus, Pg(S) and Pg(S′) are both defined.

By definition, **Pg(S) – Pg(S′) = (P – g(S)) – (P – g(S′)) = g(S′) – g(S)**.
Because S is closed the shortest known path to S′ cannot be longer than the path through S and so **g(S′) – g(S) ≤ d(S,S′)**.
Thus we see that **Pg(S) – Pg(S′) ≤ d(S,S′)**.

Because S is closed h′(S) = Pg(S). By definition h′(S′) ≥ Pg(S′). Thus, **h′(S) – h′(S′) ≤ Pg(S) – Pg(S′)**.
Combining this with the previous inequality gives the final result.

Appendix C. State Spaces Used in the Experiments

Blocks-5

There are N distinct blocks each of which is either on the "table" or on top of another block. There is a "robot" that can hold one block at a time and execute one of four operations: put the block being held onto the table, put it down on top of a specific stack of blocks, pick up a block from the table, and pick up the block on top of a specific stack. We used the 5 block version of this puzzle, which has 866 states and 2090 edges. The branching factor varies considerably from one to five, depending on the number of stacks in the state. The average branching factor is 2.4.

5-puzzle.

This is a 2×3 version of the 8-puzzle. There are 6 positions, arranged in 2 rows and 3 columns, and 5 distinct tiles, each occupying one position. The unoccupied position is regarded as containing a blank. Tiles adjacent to the unoccupied position can be moved into it, thereby moving the blank into the position just vacated. The state space comprises two unconnected regions each containing 360 states which we have connected the space by adding a single edge between one randomly chosen state in each region. Two-thirds of the states have only 2 successors, which means the branching factor at these states is effectively 1 (because every edge has an inverse, so one of the 2 successors will be the state from which the current one was reached). The other states have 3 successors.

Fool's Disk

There are 4 concentric rings with 8 integers evenly spaced around each ring. A move consists of rotating one of the rings 45 degrees clockwise or anticlockwise. Thus 8 moves are available in every state. We used the standard arrangement of integers on the disks – see [Prieditis,1993]. This gives rise to a graph containing 4096 states.

Hanoi-7

In the Towers of Hanoi puzzle there are three pegs and N different sized disks sitting on the pegs with the smaller disks above the larger disks on the same peg. The top disk on a peg may be moved onto an empty peg or onto the top of any peg whose top disk is smaller than the one being moved. We used the 7-disk version of this puzzle, which has 2187 states. Each state (except for the 3 states in which all disks are on the same peg) has 3 successors, but the effective branching factor is considerably less than 3 because of the structure of the space.

KL-2000

This is the graph "connect2000_1000.res" provided to us by Lydia Kavraki and J-C. Latombe, of Stanford University. It is produced by their algorithm for discretizing the continuous space of states/motions of robots with many degrees of freedom [Kavraki and Latombe, 1994]. This graph has 2736 nodes and an average branching factor of 10.5.

MC 60-40-7

There are M "missionaries" and C "cannibals" and a river on which there is a boat capable of holding up to B people. In any given state the boat is available on one of the river banks and a particular number of the missionaries and cannibals are on each bank. To change state, some of the people get into the boat and cross to the other side. The boat cannot change sides unless at least one person is in it. At no time, and in no place (not even the boat), may the cannibals outnumber the missionaries. We used M=60, C=40, and B=7. The resulting graph has 1878 states and an average branching factor of 20.2.

Permute-6.

A state is a permutation of the integers 1–N. There are N-1 operators numbered 2 to N. Operator K reverses the order of the first K integers in the current state. For example, applied to the state [3,2,5,6,1,7,4,...] operator 4 produces [6,5,2,3,1,7,4...]. Operator N reverses the whole permutation. We used N=6, which gives rise to 6! = 720 states. All operators are applicable in every state, so each state has 5 successors.

Words

This graph was obtained from the Stanford GraphBase which was compiled by Donald Knuth and is available in directory pub/sgb at the ftp site labrea.stanford.edu. The nodes in the graph are the 5-letter words in English. Two words are connected by an edge if they differ in exactly one letter. We use the largest connected component of this graph, which has 4493 nodes and an average branching factor of 6.