Machine Discovery of Effective Admissible Heuristics

Armand E. Prieditis
Division of Computer Science
University of California
Davis, CA 95616
(prieditis@iris.\icdavis.edu)

Abstract

Admissible heuristics are an important class of heuristics worth discovering: they guarantee shortest path solutions in search algorithms such as *A** and they guarantee less expensively produced, but boundedly longer solutions in search algorithms such as dynamic weighting. Unfortunately, effective (accurate and cheap to compute) admissible heuristics can take years for people to discover. Several researchers have suggested that abstractions of a problem can be used to generate admissible heuristics. This paper describes and evaluates an implemented program (Absolver II) that uses a means-ends analysis search control strategy to discover abstracted problems that result in effective admissible heuristics. Absolver II discovered several well-known and novel admissible heuristics, including the first known effective one for Rubik's Cube, thus concretely demonstrating that effective admissible heuristics can be tractably discovered by a machine.

1 Introduction

Admissible heuristics are an important class of heuristics worth discovering: they guarantee shortest path solutions in search algorithms such as *A** and they guarantee less expensively produced, but boundedly longer solutions in search algorithms such as *dynamic weighting* [Pohl, 1973] and *A;* [Pearl, 1984]. Unfortunately, heuristics that are both admissible and effective (accurate and cheap to compute) often take years for people to discover. For example, although the Traveling Salesperson problem was introduced in mathematical circles as early as 1931 [Lawler and Lenstra, 1984], it was not until 1971 that the Minimal Spanning Tree heuristic for it was discovered. The ultimate goal of this research is to develop a system for discovering effective admissible heuristics automatically, thereby shifting some of the burden of discovery from humans to machines. This paper describes and evaluates an implemented program (Absolver II) that can tractably discover effective admissible heuristics.

Previous proposed, but unimplemented methods to generate admissible heuristics for a problem involve finding the length of a shortest path solution to transformed version of the problem; the length is the admissible heuristic. The transformations include edge supergraphs [Guida and Somalvico, 1979; Gaschnig, 1979], operator precondition dropping [Pearl, 1984], and homomorphisms [Kibler, 1985]. Analytic results show that for such heuristics to be effective, the transformed problems that generate them should be easier to solve *and* close to the original problem [Valtorta, 1984; Mostow and Prieditis, 1989]. One suggested way to make a transformed problem easier to solve is to factor it into independently solvable subproblems when possible [Pearl, 1984]. The only implemented program to generate admissible heuristics by searching for easy-to-solve transformed problems (Absolver I) uses an exhaustive generator [Mostow and Prieditis, 1989]. Because the space of transformed problems is generally too large to search exhaustively, a smarter method of search control is required.

This paper extends previous work in three ways. First, it extends and unifies previous definitions of transformations that generate admissible heuristics (Section 2). Second, it extends Absolver I's transformation catalogs (Section 3). Finally and most important, it describes and evaluates a new search control mechanism as implemented in Absolver II (Sections 4 and 5).

2 Abstracting Transformations

Intuitively, an *abstracting* transformation removes details. To formalize this intuitive definition requires a few preliminary definitions. Let a state space search problem be a 3-tuple <S,c,P>, where S is a set of states describing situations of the world; c : S x S - - > R is a total positive cost function that returns the length of a directed arc from one state to another; and P is a predicate that characterizes a class of goal states. For example, in the Eight Puzzle problem, the set of states consists of all tile permutations; the cost function on a pair of states returns 1 if one state is reachable from the other by swapping the blank with an adjacent tile, and oo otherwise. A goal predicate might specify that the tiles are to be in a particular order. Problem solving involves finding a finite sequence of states leading from the initial state to a state that satisfies the goal predi-

cate. The cost function can be specified explicitly with a matrix or implicitly, relative to a set of parameterized actions called *operators*. Similarly, the goal predicate can be specified explicitly by enumerating all goal states or implicitly, relative to a goal *statement* that defines a class of states.

A function $\phi : S \rightarrow S'$ is *abstracting* from problem $<S, c, P>$ to problem $<S', c', P'>$ iff for all $s, t \in S$:

1. **ϕ is *total*: $\phi(s) \in S'$**
2. **ϕ *preserves or reduces cost*: $c'(\phi(s), \phi(t)) \leq c(s, t)$**
3. **ϕ *preserves goals*: $P(s) \Rightarrow P'(\phi(s))$**

As shown in Figure 1, a heuristic for a state in problem $<S, c, P>$ is computed by first computing the state and then finding a shortest path solution in the abstracted problem $<S', c', P'>$ (e.g. with *A**). The length of that shortest path solution is the admissible heuristic.[1] For example, if the requirement that the adjacent location need be blank is dropped from moves in the Eight Puzzle problem, then moves in the abstracted problem will result in states where tiles are superimposed. The length of a shortest path solution in this transformed space is therefore an admissible heuristic for the original problem. In fact, this length is the Manhattan Distance over summed all tiles (i.e. the rectilinear distance to each tile's goal destination).

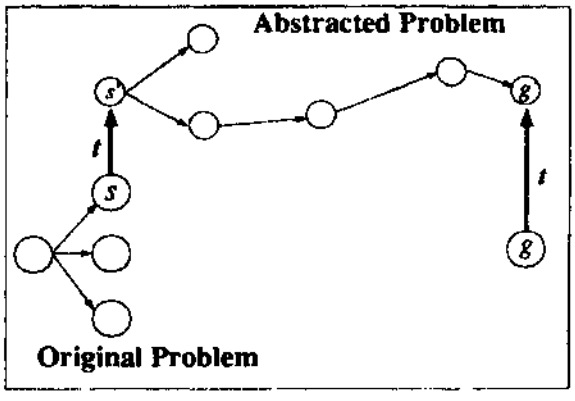

Figure 1: How a Heuristic is Computed in Our Model

Our definition of abstraction is sufficiently general to cover edge supergraph transformations, precondition dropping transformations, homomorphisms, and other admissible-heuristic-generating transformations not covered by these (e.g. subgoal dropping).

3 Extended Transformation Catalogs

Table 1 shows Absolver IPs catalog of abstracting transformations, which operate on the same STRIPS-style problem representation [Fikes *et al,* 1972] of Absolver I. Each transformation has been proved to be abstracting [Prieditis, 1990], thereby guaranteeing that all heuristics generated from the catalog are indeed admissible. Since using breadth-first search to compute a heuristic generated from an abstracted problem is generally too expensive, we have implemented a catalog of speedup transformations (shown in Table 2 and henceforth called *speedups),* each of which has been proved to preserve admissibility [Prieditis, 1990].

Type	Name	English Paraphrase
Information Dropping	drop_pre(p,o)	drop precondition p of op o
	drop_goal(p)	drop p from goal
	drop(p)	drop p everywhere
Mapping	count(p)	replace p by number of p(x)
	parity(i)	replace i^{th} integer by its parity
Composition	sum(i,j)	replace two integers by their sum
	bagsum(i,j)*	replace two bags by their union

Table 1: Our Current Catalog of Abstracting Transformations (* = new)

Name	English Paraphrase
Remove Subsumed*	remove an operator that applies less often and results in the same state as another operator
Remove Irrelevant*	remove an operator that cannot be on a shortest solution path
Factor	factor a problem into independent subproblems
Finite Differencing*	incrementally update the heuristic as a result of a base-level move
Precompute Lookup Table*	store the distance from goal to every reachable state
Collapse to Closed Form	collapse non-branching search to closed form of shortest path length

Table 2: Our Current Catalog of Speedup Transformations (* = new)

For example, after applying drop(Blank) in the Eight Puzzle, the resulting abstracted problem can be factored into a set of independently solvable problems; then finite differencing can be applied to recompute only that factor's heuristic that requires updating after each move in the original space; and finally, a lookup table can be precompiled for each factor. For *n* x *n* puzzles, the complexity reduction is as follows (original problem is leftmost):

$$O(n!) \overset{\text{abstract}}{\Longrightarrow} O(n^{2n^2-2}) \overset{\text{factor}}{\Longrightarrow} O(n^4) \overset{\text{finite differencing}}{\Longrightarrow} O(n^2) \overset{\text{precompute}}{\Longrightarrow} O(1)$$

Figure 2 summarizes our model for discovering admissible heuristics. An abstracted version of the original problem is sped up and then used as an admissible heuristic for all problem instances (problem + initial state). Notice that the effort to discover a heuristic for a problem is amortized over all instances of a problem because the derived heuristic is not instance-specific.

4 Absolver II

Absolver II uses a table (Table 3) of plausible abstracting transformations for each implemented speedup to identify and eliminate obstacles to applying speedups.[2] An "x" in a particular row/column entry of this table means that the row's abstracting transformation is likely to lead to satisfying the column's speedup transformation, given our experience in applying the model by hand.

[1] Admissibility, composability, partial ordering, and other results are proved in [Prieditis, 1990].

[2] Finite Differencing is not shown because it is implicitly part of Factor; Precompute is not shown because it depends on hard-to-predict information such as search space size.

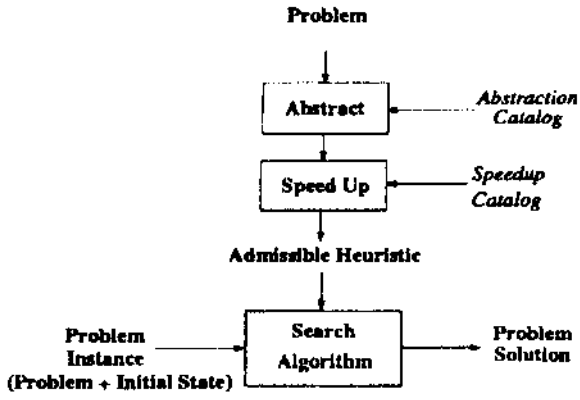

Figure 2: Our Model for Discovering Admissible Heuristics

	Speedup Transformation			
Abstraction	Factor	Remove Subsumed	Remove Irrelevant	Collapse to Closed Form
drop-pre(p,o)	x		x	
drop-goal(p)	x		x	
drop(p)	x		x	
sum(i,j)		x		x
bagsum(i,j)		x		x
count(p)		x		x
parity(i)		x		

Table 3: Plausible Abstractions for Each Speedup Transformation

Using this table, Absolver II outputs the first heuristic it finds, which it assumes to be the best one, subject to the following evaluation criteria implicit in its search mechanism:

- The less information dropped to derive the heuristic the better (e.g. drop fewer preconditions).
- A heuristic must be derived from an abstracted problem that can be sped up. To avoid generating useless heuristics, Absolver II incorporates necessary conditions for speedup into its generator of abstractions.

Absolver II is actually comprised of three subprograms: Composer, Dropper, and Summarizer. Using several meta-heuristics, Composer searches through the space of composition abstractions (e.g. sum) for those abstractions that lead to the removal of subsumed operators. Dropper attempts to factor a problem into independent subproblems by applying drop-pre and drop-goal (irrelevant operators are implicitly removed; drop is modeled in terms of these two transformations). Dropper first builds a set of preconditions and subgoals to drop that are sufficient to make a pair of subgoals independent. Next, using several meta-heuristics and a standard search algorithm for combining these drops, it tries to find a minimal set of drops that allows factoring. Summarizer attempts to obtain a problem in which subsumed operators can be removed by applying count (to every predicate). Failing that, Summarizer applies parity (to every integer) in an attempt to obtain a problem in which subsumed operators can be removed.[3]

Absolver IPs overall search strategy is as follows. First, it calls Composer. If Composer fails to find a problem in which a certain percentage of subsumed operators can be collapsed, then Absolver II calls Dropper. If Dropper cannot factor the problem into two or more independent subproblems, then Absolver II calls Summarizer as a last resort. After Absolver II succeeds in applying a speedup transformation within either Composer, Dropper, or Summarizer, it calls itself recursively on the resulting problem, thus resulting in a hierarchy of abstractions, which generates a hierarchy of heuristics, each of which is used to more efficiently compute heuristics lower in the hierarchy. The rest of this section describes Composer and Dropper; Summarizer is not described since it is relatively straightforward.

4.1 Composer: The Search for Subsumability

Composer's goal is to find an abstracted problem in which subsumed operators can be removed. It tries to reach this goal by searching through the space of pair-wise compositions with a standard hill-climbing algorithm and a meta-heuristic in the form of a "similarity" coefficient on each candidate composition. The similarity coefficient returns the number of pairs of operators in which the composition results in the same value. The larger the similarity coefficient, the more likely that a particular composition will lead to operator subsumption. Composer applies the similarity coefficient to each pair-wise candidate composition and then proceeds in the direction of that pair-wise composition with the largest similarity coefficient (ties are broken arbitrariiy)-

When the number of uninstantiated operators after subsumed operators are removed is 75% of the number of original operators, Composer calls itself recursively to build a hierarchy of heuristics. The 75% value, which we chose initially and have not had to change, is a rough indicator that the branching factor of the abstracted problem is sufficiently reduced such that search will be cheaper than in the original space, but not reduced so much that inaccurate though cheap-to-compute heuristics result.

For example, in the Fool's Disk problem (shown in Figure 3), where the object is to rotate each of the concentric disks until the numbers on each radius (labeled r_1-r_s) sum to 12, Composer generates the hierarchy of problems shown in Figure 4, each of which generates an admissible heuristic.

The Diameters problem is a transformation of the original Fool's Disk problem where the following pairs of radii are summed using the sum transformation: r_1 and r_5, r_2 and r_6, r_3 and r_7, and r_4 and r_8. The Perpendicular Diameters problem is a transformation of the Diameters problem where perpendicular diameters are summed. For example, in the Perpendicular Diameters problem, the composite r_1 and r_5 is summed with the composite r_3 and r_7. The All Numbers problem is a transformation of the Perpendicular Diameters problem:

[3] Currently, Absolver II collapses problems to closed form only opportunistically.

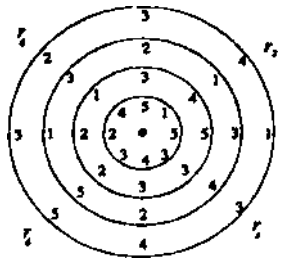

Figure 3: The Fool's Disk Problem

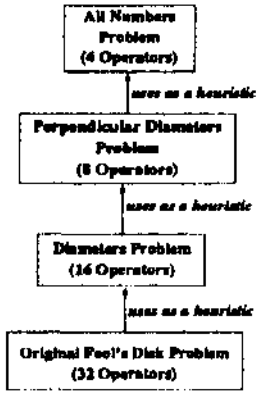

Figure 4: The Hierarchy of Discovered Heuristics for the Fool's Disk Problem

all the numbers are summed. To compute, for example, the Diameters problem heuristic for any state in the original Fool's Disk problem, the state is abstracted (by summing opposite radii) and then the search algorithm (e.g. *A**) is called recursively with the Diameters problem and the abstracted state. If this heuristic returns oo for a state at any level of abstraction, then the state can be pruned. In sum, to prune states that cannot reach the goal, the original Fool's Disk relies on the Diameters problem, which in turn relies on the Perpendicular Diameters problem, which finally relies on the All Numbers problem.[4]

The complexity of Composer is dominated by the complexity of computing the similarity coefficient, whose complexity is $O(m^2)$ for m operators. For each nonrecursive call of Composer, the similarity coefficient will be computed $n(n-1)/2$ times for n integers since only integer pairs (or bags) are processed. Thus the total complexity of Composer is $O(m^2n^3)$ since Composer will call itself recursively at most *0(n)* times.

4.2 Dropper: The Search for Factorability

Dropper is actually composed of two subprograms: Pairwise and Combine. Over each pair of subgoals, Pairwise finds a set of precondition and subgoal drops sufficient to make the pair of subgoals independent. For example, Figure 5 shows that given a subgoal At(I,A) and a subgoal At(2,B) in the Eight Puzzle, an operator of the form Move(1,-,A) directly adds At(I,A) and an operator of the form Move(2,-,B) directly adds At(2,B). Since Moved,.,A) places the blank in some location (possibly location B) and Move(2,_,B) places the blank in some location (possibly location A), these operators *might* interact through Blank of both operator's preconditions. Dropping Blank from the precondition of the Move operator eliminates this interaction. To test if the entire goal is factorable into at least two parts, the drop sets from each pair of subgoals must be combined somehow. Combine uses an iterative deepening search algorithm with the meta-heuristic of focusing on the most frequently occurring drops first. This search algorithm tends to minimize the number of drops to obtain a problem that can be factored into two or more subproblems. In our Eight Puzzle example, which only has a single drop set (dropping Blank from the operator and the goal), Combine converges to a factorable set after one iteration. Combine is then called recursively on each of the factors. It then terminates successfully since no other factorings can be obtained within each factor and produces the AND tree of factors shown in Figure 6, which is used to compute the Manhattan Distance heuristic.

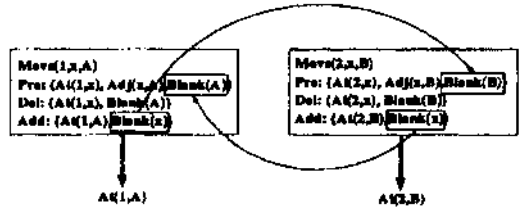

Figure 5: Dropping the Blank Enables Factoring (Arrows are Potential Interactions)

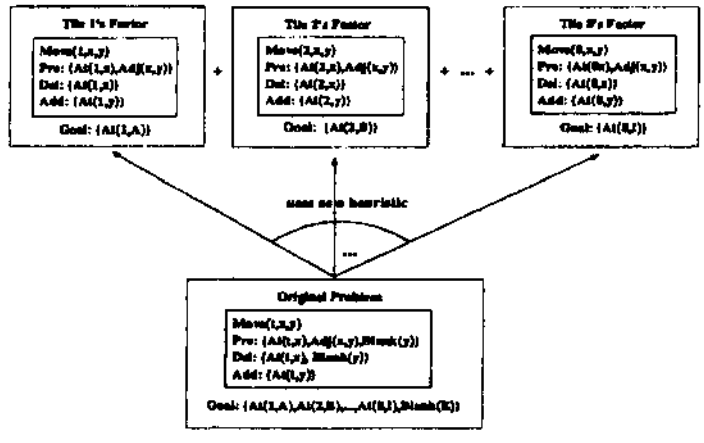

Figure 6: Independently Solvable Factors Used to Compute Manhattan Distance

The complexity of Dropper is dominated by Pairwise,

[4] These admissible heuristics are not to be confused with the Fool's Disk problem-solving *strategy* described in [Ernst and Goldstein, 1982].

which constructs a transitive closure of the set of operators that can possibly lead to achieving a goal. The worst-case complexity of constructing this transitive closure is $O(b_b^{d_b})$, where b_b is the *backward branching factor* (the average number of operator instances that adds a given predicate) and d_b is the *backward depth* (the length of the longest chain of operators in the set of relevant operators for a given subgoal). That is, the number of preconditions that Pairwise examines is proportional to the number of paths from a subgoal to each operator in the chain.

5 Experimental Results

Table 4 presents the results of applying Absolver II to several well-defined search domains, each of which is sufficiently complex to require heuristics. The table lists the domain, the name of the heuristic, and the percentage of the space that was explored (to two significant digits). The percentage is computed by dividing the number of heuristics generated before the named one was found by the number of abstractions with respect to our catalog of abstracting transformations and multiplying by 100.[5] The space size is a conservative estimate in that it includes only those abstractions that Absolver II actually used. For example, the space size for deriving the Manhattan Distance is 4096, since Absolver II only drops operator preconditions and subgoals to derive the heuristic and there are 9 subgoals and 3 preconditions (2^9+^3 = 4096). Since integers are not part of the problem specification for the Eight Puzzle, Absolver II does not apply transformations such as sum. In contrast, to derive the Fool's Disk heuristic, Absolver II only applies sum (to the 8 radii) and not other transformations; it therefore searches a space of size 2^8 (= 256).

The results of this table can be summarized as follows. Absolver II discovered effective admissible heuristics in 6 out of the 13 domains by exploring only a fraction of the space of heuristics derivable by the abstracting transformations in our catalog. Absolver II discovered 8 novel heuristics, 5 of which turned out to be effective. The novel effective heuristics include the first known (nontrivial) admissible heuristic for the 3 x 3 x 3 Rubik's Cube (partially precomputed Center-Corner), which resulted in roughly 8 orders of magnitude speedup with *A** over blind search for long solutions; the best admissible heuristic for the Eight Puzzle (precomputed X-Y), which expanded 1.8 times fewer states than the Linear Conflict heuristic [Hansson *et al.*, 1989], an adjusted more informed version of the Manhattan Distance heuristic; a heuristic in the Rooms World problem (search-computed Box Distance), which counts the minimum number of rooms each box must pass through to reach its goal destination and which expanded 267 times fewer states than blind search; the Diameters Fool's Disk heuristic, which expanded 45.41 times fewer states than blind search; and an Instance Insanity heuristic, which is analogous to the Diameter's heuristic in that opposite side colors are combined and which expanded 2.61 times fewer states than blind search.

Domain	Heuristic	% Space Explored
Eight Puzzle	Manhattan Distance+	.0024
	X-Y*+	.000045
TSP	Unvisited Cities	.78
Towers of Hanoi	# Misplaced Disks	.00038
Mutilated Checkerboard	Colored Squares+	25
2-D Routing	Unvisited Signals*	.0097
Rubik's Cube	Center-Corner*+	10^{-15}
Fool's Disk	Diameters*+	2.7
Instant Insanity	Nearly Opposite Sides*+	18
Think-A-Dot	Dropped Gates*	2.7×10^{-5}
Rooms World	Box Distance*+	25
Blocks World	# Misplaced Blocks	.0025
Eight Queens	# Unplaced Queens	.2
Uniprocessor Scheduling	Unassigned Jobs*	8.7×10^{-8}

Table 4: Table of Admissible Heuristics Discovered by Absolver II (* = novel; + = effective)

Absolver II derived several known admissible heuristics including the Manhattan Distance heuristic of the Eight Puzzle (using a different formulation than for the X-Y heuristic), the Number of Misplaced Disks heuristic of the Towers of Hanoi, the Mutilated Checkerboard heuristic, and the Number of Misplaced Blocks heuristic. All except the Manhattan Distance heuristic were of the same complexity as the originals—the Manhattan Distance heuristic is slower by $O(n)$ for n x n puzzles because the derived heuristic uses search to compute the number of moves needed to get each tile from its current location to its goal location.[6]

Absolver II derived several inferior heuristics. In the 2-D Routing domain, Absolver II derived the "Unvisited Signals" heuristic, which returns a count of the number of non-reached signal locations, instead of the more informed Steiner Tree heuristic, which returns the length of a minimum rectilinear spanning tree and which we derived by hand [Prieditis, 1990], because it dropped a relation that was too salient for the problem. In the Traveling Salesperson Problem, Absolver II derived the "Unvisited Cities* heuristic, which computes the sum of the least cost edges leading to an unvisited city over all unvisited cities, instead of the more informed Minimal Spanning Tree heuristic, which we derived by hand [Prieditis, 1990], because we have not implemented the speedup required to derive the Minimal Spanning Tree heuristic.

Absolver II failed to find an effective admissible heuristic for the Think-A-Dot, Eight Queens, and Scheduling problems for the same reason we failed to find one by

[5] Problem formulations, methods for choosing good formulations, derivations, and performance of the resulting heuristics are detailed in [Prieditis, 1990].

[6] Assuming finite differencing and ignoring a one time $O(n^4)$ initial computation.

hand [Prieditis, 1990]: all abstractions that could be sped up removed too many important details and therefore resulted in relatively uninformed heuristics. For example, to obtain the factorable abstracted problem that results in the Unassigned Jobs heuristic for the Scheduling domain, the time and seriality constraints must be dropped. The resulting heuristic, which returns the minimum costs of unassigned jobs, is poorly informed. Our approach to discovering admissible heuristics appears to be unsuitable when the original problem is characterized by high goal or operator "interference" (many operators directly change a large proportion of subgoals in the original problem or make a large proportion of operators subsequently inapplicable) and all abstracted problems are characterized by low goal or operator "interference" (few operators directly change a large proportion of subgoals or make operators inapplicable). Of course, it may be that to derive effective admissible heuristics for such domains requires abstractions and speedups beyond our model or that we were not able to find a "good" formulation in which to derive heuristics.

6 Conclusions and Future Work

Absolver II tractably discovered many known admissible heuristics (though with varying efficiency relative to the original versions) and some novel effective admissible heuristics by using a means-ends analysis search strategy. Some of these heuristics were still effective even though they are computed by search. Such search-computed heuristics might be important in domains where effective closed-form heuristics cannot be found (e.g. Rooms World).

As Absolver II is an experimental system, it has several shortcomings, each of which suggests interesting directions for future research. First, because it sometimes drops salient relations of a problem, it might be enhanced by a theory that links information loss via abstraction to accuracy of the resulting heuristics. This theory might allow Absolver II to predict the effectiveness of heuristics without testing them, which is currently left up to the user. Second, non-abstracting transformations might be required to derive effective heuristics in certain domains (e.g. Eight-Queens). Third, Absolver II might be able to boost the informedness of certain admissible heuristics by taking into account interactions in the base level between independently solvable factored subproblerns in the abstract level. For example, the LC heuristic might be derivable by incrementing the Manhattan Distance heuristic by 1 for each base-level interaction found between independent factors with only one shortest path solution. Finally, Absolver II sometimes derives inferior heuristics because it overestimates the number of relations to drop to remove interactions; examples could help it focus on those interactions that actually occur. For example, Absolver II could assume (at the cost of admissibility) that a problem is factorable until an interaction is detected among its factors when computing a heuristic for a particular problem instance. Once such an interaction is detected, its cause could be located and eliminated by abstraction.

Despite its shortcomings, Absolver II concretely demonstrates that effective admissible heuristics can be tractably discovered by a machine.

Acknowledgements

Thanks go to Jack Mostow, Tom Mitchell, Alex Borgida, Saul Amarel, Haym Hirsh, Christina Chang, Mukesh Dalai, Sridhar Mahadcvan, and Prasad Tadepalli. Thanks also go to Rich Cooperman for testing the Rubik's Cube heuristics and to Rich Korf for supplying the *IDA** program used to collect data for the Manhattan Distance heuristic.

References

[Ernst and Goldstein, 1982] G. Ernst and M. Goldstein. Mechanical discovery of classes of problem-solving strategies. *J A CM,* 29(1):1 23, 1982.

[Fikes *et al.,* 1972] R. Fikes, P. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence,* 3(4):251-288, 1972. Also in *Readings in Artificial Intelligence,* Webber, B. L. and Nilsson, N. J., (Eds.).

[Gaschnig, 1979] J. Gaschnig. A problem-similarity approach to devising heuristics. In *Proceedings IJCAI-6,* pages 301 307, Tokyo, Japan, 1979. International Joint Conferences on Artificial Intelligence.

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

[Hansson *et al.,* 1989] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics, 1989. Unpublished document: forwarded by the authors.

[Kibler, 1985] D. Kibler. Natural generation of heuristics by transforming the problem representation. Technical Report TR-85-20, Computer Science Department, UC-Irvine, 1985.

[Lawler and Lenstra, 1984] E. Lawler and L. Lenstra. *The Traveling Salesman Problem.* John Wiley and Sons, New York, 1984.

[Mostow and Prieditis, 1989] J. Mostow and A. Prieditis. Discovering admissible heuristics by abstracting and optimizing. In *Proceedings IICAI-11,* Detroit, MI, August 1989. International Joint Conferences on Artificial Intelligence.

[Pearl, 1984] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem-Solving.* Addison-Wesley, Reading, MA, 1984.

[Pohl, 1973] I. Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings IJCAI-3,* pages 20-23, Stanford, CA, August 1973. International Joint Conferences on Artificial Intelligence.

[Prieditis, 1990] A. Prieditis. *Discovering Effective Admissible Heuristics by Abstraction and Speedup: A Transformational Approach.* PhD thesis, Rutgers University, 1990.

[Valtorta, 1984] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences,* 34:47-59, 1984.