Machine Learning, 12, 117-141 (1993)
© 1993 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Machine Discovery of Effective Admissible Heuristics

ARMAND E. PRIEDITIS PRIEDITIS@cs.UCDAVIS.EDU
Department of Computer Science, University of California, Davis, CA 95616

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 certain transformations of a problem can be used to generate admissible heuristics. This article defines a more general class of transformations, called *abstractions*, that are guaranteed to generate only admissible heuristics. It also 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 a few 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.

Keywords. Machine discovery, admissible heuristics, search, abstraction.

1. Introduction

Admissible (lower-bound) heuristics are an important class of heuristics worth discovering because they have several desirable properties in various search algorithms. For example, they guarantee shortest path solutions in the A^* algorithm. The computational complexity of generating a shortest path solution, which is NP-Complete for many problems, is sometimes justified when path length corresponds to resource usage such as time or money or when the same solution is used frequently. Often, however, a reasonably short solution for a problem is acceptable if it can be generated faster. With admissible heuristics, the *dynamic weighting* (Pohl, 1973) and A_ϵ^* (Pearl, 1984) algorithms guarantee less expensively produced, but boundedly longer solutions. Moreover, it is possible to reduce an exponential average time complexity to a polynomial one using A^* and multiples of an admissible heuristic (Chenoweth & Davis, 1991). 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 & Lenstra, 1984), it is not until 1971 that the Minimal Spanning Tree heuristic for it was discovered (Held & Karp, 1970). 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 article 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 a transformed version of the problem. As shown in figure 1, a heuristic for a state s in problem with goal g is computed

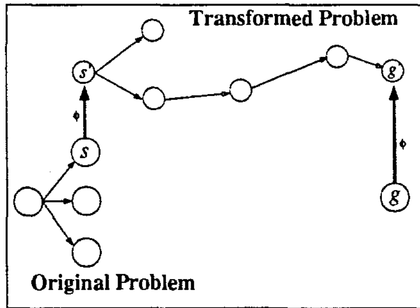

Figure 1. The length of a shortest path of a transformed problem = The admissible heuristic.

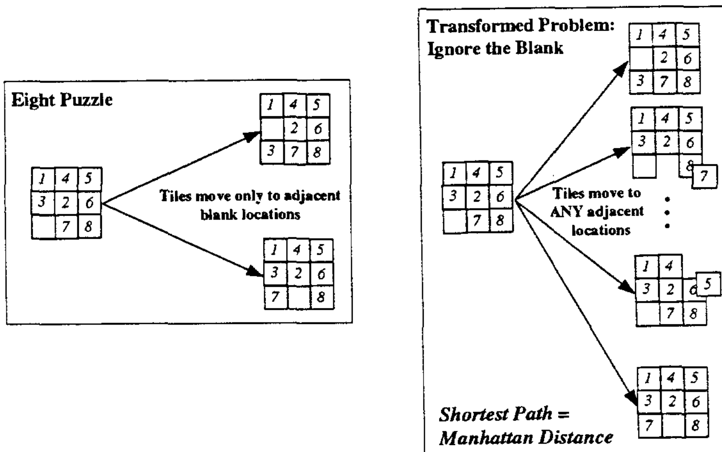

Figure 2. How Manhattan Distance is generated from a transformed problem.

by transforming s to s' and g to g', and then finding a shortest path solution from s' to g'; the *length* of that shortest path solution is the admissible heuristic. For example, figure 2 shows how the Manhattan Distance heuristic, an admisible heuristic for sliding block puzzles, can be generated by ignoring the blank. Since moves in the transformed problem will result in states where tiles are superimposed, the length of a shortest path solution in this transformed space is the sum over all tiles of the rectilinear distance to each tile's goal destination. This sum, which underestimates the actual solution path length because it allows tiles to be superimposed, is the Manhattan Distance.

Transformations that generate admissible heuristics include adding edges to a problem's search graph (Guida & Somalvico, 1979; Gaschnig, 1979), dropping operator preconditions (Pearl, 1984), and applying homomorphic transformations to a problem (Kibler, 1985). The intuitive reason that these transformations generate admissible heuristics is because they add short-cut solution paths.

For such heuristics to be effective, the transformed problems that generate them should be easier to solve *and* closely approximate the original problem (Valtorta, 1984; Mostow

and Prieditis, 1989). Several methods exist to make a problem easier to solve. These methods include factoring the problem into independently solvable subproblems (Pearl, 1984) or applying an efficient solution-producing algorithm when possible. Absolver I exhaustively searches for such easy-to-solve transformed problems (Mostow & Prieditis, 1989). One major problem with Absolver I is that for most problems the space of transformed problems is generally too large to search exhaustively—a more efficient method of search control is required.

This article 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) and compares this mechanism to other machine discovery systems (section 6).

2. Abstracting transformations

Intuitively, an *abstracting* transformation removes details. To formalize this intuitive notion requires a definition of search. The definition that we will assume is standard in the AI literature (Nilsson, 1980). Search can be thought of as finding a finite path in a graph from a node representing an initial state (situation) to a node that satisfies a given goal. Certain pairs of nodes are connected by directed arcs that represent the application of an operator; these arcs are typically weighted to represent the cost of applying the corresponding operator. The graph and goal can be specified explicitly or implicitly. In an explicit specification, the nodes and arcs with associated costs might be supplied in a table that includes every node in the graph and a list of its successors and the costs of associated arcs. This information might also be specified by a matrix that stores the costs of associated arcs for every pair of nodes (an infinite cost arc represents the absence of an arc). Similarly, the goal might be specified by enumerating all goal states. In an implicit specification, only that portion of the graph that is sufficient to include a goal node is made explicit by applying operators using a search algorithm such as A^*. In this type of specification, the set of goal states is specified by supplying a goal statement that defines that class of goal states. For example, in the Eight Puzzle problem, the set of states consists of all tile permutations, and operators only allow swapping the blank with an adjacent tile (i.e., the cost function on a pair of states return 1 if one state is reachable from the other by swapping the blank with an adjacent tile, and ∞ otherwise). The set of goal states might contain all those states with tiles in a particular order.

More formally, let a search problem be a 3-tuple $\langle S, c, G \rangle$, where S is a set of states describing situations of the world; $c : S \times S \rightarrow \mathbb{R}$ is a positive cost function that represents the cost of applying the corresponding action from one state to another; and $G \subseteq S$ is a set of goal states. The reason for defining a search problem to exclude the initial state will be explained shortly. A heuristic evaluation function for a problem $\langle S, c, G \rangle$ is a positive real-valued function that estimates the cost of a minimum cost path to the goal node.

Using this framework, several standard functions can now be defined. The length of a shortest path from a state s to a state t for a set of states S with cost function c is defined as the function $k^*_{S,c}(s, t)$:

$$k^*_{S,c}(s, t) = \min \left\{ \sum_{i=1}^{n-1} c(s_i, s_{i+1}) \mid s_1 = s \bigwedge_{1 \leq i \leq n} s_i \in S \wedge s_n = t \right\}$$

The length of a shortest path from a state s to a state in G of a problem $\langle S, c, G \rangle$ is defined as the function $h^*_{S,c,G}(s)$:

$$h^*_{S,c,G}(s) = \min \{k^*_{S,c}(s, t) \mid t \in G\}$$

The reason that the initial state is excluded from our definition of a search problem is so that the definition for the length of a shortest path is consistent with the notion of a heuristic in standard AI texts (Nilsson, 1980), which is also relative to a search problem $\langle S, c, G \rangle$ rather than a search problem plus an initial state.

An admissible heuristic for a problem $\langle S, c, G \rangle$ is a function $h : S \to \mathbb{R}$ such that for all $s \in S : h(s) \leq h^*_{S,c,G}(s)$.

A function $\phi : S \to S'$ is *abstracting* from problem $\langle S, c, G \rangle$ to problem $\langle S', c', G' \rangle$ iff:

1. ϕ *reduces cost*: $(\forall s, t \in S) c'(\phi(s), \phi(t)) \leq c(s, t)$
2. ϕ *expands goals*: $(\forall g \in G) \phi(g) \in G'$

An example of an abstracting transformation is one that ignores the blank in our Eight Puzzle problem where $\langle S, c, G \rangle$ is the original problem and $\langle S', c', G' \rangle$ is the transformed problem such that S' is the set of all tile situations with superimpositions allowed, $c'(s, t) = 1$ when s can be changed into t by moving a tile into an adjacent location and ∞ otherwise (i.e., the operators that define the cost function for the transformed problem allow moving a tile to an adjacent, possibly non-blank location), and $G' = G$ but with the blank ignored. The transformation is abstracting because it reduces cost and expands goals from the original to the transformed problem.

Abstracting transformations have several important properties in the context of search, each of which is proved by Prieditis (1990). First, they generate admissible heuristics: if $\phi : S \to S'$ is abstracting from problem $\langle S, c, G \rangle$ to problem $\langle S', c', G' \rangle$, then $(\forall s \in S) h^*_{S'c'G'}(\phi(s)) \leq h^*_{S,c,G}(s)$. Second, they are composable: if $\phi_1 : S_1 \to S_1'$ is abstracting from problem $\langle S_1, c_1, G_1 \rangle$ to problem $\langle S_1', c_1', G_1' \rangle$, $\phi_2 : S_2 \to S_2'$ is abstracting from problem $\langle S_2, c_2, G_2 \rangle$ to problem $\langle S_2', c_2', G_2' \rangle$, $S_1' \subseteq S_2$, $(\forall s, t \in S_1') c_2(s, t) \leq c_1'(s, t)$, and $G_1' \subseteq G_2$, then $g \circ f : S_1 \to S_2'$ is abstracting from problem $\langle S_1, c_1, G_1 \rangle$ to problem $\langle S_2', c_2', G_2' \rangle$.

Third, two search problems can be partially ordered according to whether a composition of abstracting transformations exists to get from one to the other. This partial order relation makes it easy to understand why certain heuristics always dominate others in terms of pruning power. For example, the Number of Out-of-Place Tiles heuristic, which is less accurate than the Manhattan Distance heuristic, is generated by additionally ignoring the adjacency

requirement. Because abstracting transformation remove details, the more abstract the problem, the less accurate the resulting heuristic. Since accuracy determines pruning power (Nilsson, 1980), generally the more abstract a problem the lower the pruning power of the resulting heuristic.

Finally, for every admissible heuristic, an abstracting transformation that generates a heuristic at least as accurate as the original heuristic can be constructed. More formally, given an admissible heuristic $h(s)$ for a problem $\langle S, c, G\rangle$, an abstracting transformation $\phi : S \rightarrow S'$ from problem $\langle S, c, G\rangle$ to problem $\langle S', c', G'\rangle$ that generates a heuristic at least as accurate as $h(s)$ can be constructed as follows. Let $\phi(s) = s$, $S' = S$, $G' = G$ and for all $s, t \in S$:

$$c'(s, t) = \begin{cases} h(s) & \text{if } t \in G \\ c(s, t) & \text{otherwise} \end{cases}$$

Clearly, ϕ is abstracting and $h^*_{S',c',G'}(\phi(s))$ is at least as accurate as $h(s)$.

Abstracting transformations are sufficiently general to cover previous definitions of transformations that generate admissible heuristics, including adding edges (Guida & Somalvico, 1979; Gaschnig, 1979), dropping operator preconditions (Pearl, 1984), and applying homomorphic transformations (Kibler, 1985). For example, adding an edge to a problem is the same as reducing the cost function from ∞ to some finite value for the pair of states bridged by that edge. Abstracting transformations also cover other transformations not covered by previous definitions. For example, the abstracting transformation of dropping a conjunct from a conjunctive goal description generates an admissible heuristic because it increases the set of goal states. However, this transformation is not covered by adding edges, dropping operator preconditions, or applying a homomorphic transformation.

3. Extended transformation catalogs

Absolver II's abstracting and speedup transformations operate on the same extended STRIPS-style problem representation (Fikes et al., 1972) of Absolver I. A state in this representation is a set of *ground literals*; the set of goal states is specified implicitly by a *goal statement*, which specifies those ground literals that must be in a goal state; and the cost function is specified relative to a set of parameterized operators—if an operator in this set applies to a state, the cost of getting from that state to the state that results from applying the operator is 1; otherwise the cost is ∞. As in the original STRIPS, an operator contains a *precondition* set, which specifies the set of literals that must be in the state before the operator can be applied to that state, a *delete* set, which specifies the set of literals that will no longer be in the state after the operator is applied, and an *add* set, which specifies the set of literals that will be in the state after the operator is applied.

We have extended the original STRIPS problem representation to include *integer* arrays, *bit* arrays, and arrays containing *bags* of propositional items (a bag, also known as a multiset, is a set with duplicates allowed). The only operations on these items are the following: add a constant (integers), invert a bit (bits), and add to a bag (bags). Each operator is augmented

with three arrays (one for each array type), each of which specifies how each array element changes with operator application. These arrays are called the *delta constant* arrays. The only tests on array items are the following: equal to a constant (integers), equal to 0 or to 1 (bits), and equal to a particular bag (bags); when no such test is included for an array item, the test is simply ignored for that item (i.e., a "don't care" condition). Each operator is augmented with three such arrays for precondition tests—one for each array type. In addition to satisfying an operator's precondition set before the operator can be applied, the preconditions of every element in each array type must be satisfied. The goal statement is similarly augmented with three such arrays. A state, which consists only of literals in the original STRIPS representation, is augmented to include the three array types to store the current values for every element of each array type. Since the actual *implementation* of operators, goal statements, and states is not important for purposes of this article, we will not describe it in detail; what is important is that objects such as literals and integers exist in our representation and can be manipulated. We chose this representation because its declaratively represented operators, goal, and states make it easy to compute the form of an abstracted operator, goal, and state, and because it is powerful enough to represent a wide variety of search problems.

Table 1 summarizes Absolver II's catalog of abstracting transformations, each of which has been proved to be abstracting (Prieditis, 1990), thereby guaranteeing that all heuristics generated from the catalog are indeed admissible. Prior to problem-solving, these transformations are applied to operators and/or the goal statement to build an abstracted problem. During problem-solving, the transformations are applied to those states that are actually reached during search. We have categorized the transformations into three distinct types: *information-dropping* which simply remove a piece of information from an operator or a goal; *mapping*, which map one or more objects in the original space to one object in the abstracted space; and *composition*, which functionally compose two pieces of information. For example, `sum(i, j)` is a composition abstraction because it takes two integer items

Table 1. Absolver II's current catalog of abstraction transformations over operators, goal statements, and states.

Type	Name	English paraphrase
Information dropping	`drop_pre(p,o)`	drop precondition `p` from operator `o`
	`drop_goal(p)`	drop `p` from the goal statement
	`drop(p)`	drop `p` from operators, states, and the goal statement
Mapping	`count(p)`	replace `p` by number of `p(x)` in operators, states, and the goal statement
	`parity(i)`	replace i^{th} element of the integer array by its parity in operators, states, and the goal statement
Composition	`sum(i,j)`	replace two integers elements by their sum in operators, states, and the goal statement
	`bagsum(i,j)*`	replace two bags by their union in operators, states, and the goal statement

** = new: not in Absolver I.

in an integer array and replaces them by their sum, which is a functional composition of the original items. Each operator is transformed as follows: if the precondition test for either element is a "don't care," the test for the composition becomes a "don't care"; otherwise, the constants for each test are added to form the test for composition. In either case, the original two elements are removed and a new array element representing the composition is added. The goal statement is similarly transformed. The elements `i` and `j` are removed from each operator's delta constants array and summed to form a delta constant for the new array element.

Although we do not claim that this catalog is complete or "right" for our representation, we did choose each transformation for several reasons. First, each transformation is *domain-independent*, which implies that it will apply across different domains. Second, each transformation is *mechanizable*—that is, its application can be automated (and in fact is). Third, each transformation appeared *well motivated*—we had examples of known heuristics that could be derived with it. Finally, each transformation is somewhat *natural* for our representation: it can be concisely and compactly defined.

As we mentioned in section 1, an abstracted problem that generates a heuristic should be easier to solve than the original problem. Since using breadth-first search to compute a heuristic generated from an abstracted problem is generally too expensive, some method to speed up the computation of a heuristic is required. Towards that end, we have implemented the catalog of speedup transformations (henceforth called *speedups*). Table 2 lists these speedups from least to most powerful in terms of reducing the complexity of search. We have also proved that each of the speedups preserves admissibility (Prieditis, 1990).

Removing a redundant or an irrelevant operator reduces the branching factor. Since the form of our operators is declarative, the test for redundancy is relatively straightforward: an operator is redundant if its precondition, add, and delete sets and its delta constant arrays exactly match that of another operator (modulo different names for the same parameter). The test for irrelevancy is, however, somewhat more complicated. An operator is *relevant* for a goal if it can directly add a conjunct in the goal statement or if it can add a conjunct in the precondition of a relevant operator. If an operator is not in this set, it is irrelevant and can be removed. An operator is actually "removed" by specializing its parameters. For example, the parameterized operator `Move(t,x,y)`, which moves tile `t` from location

Table 2. Absolver II's current catalog of speedup transformations.

Name	English paraphrase
Remove Redundant*	remove a redundant operator
Remove irrelevant*	remove an operator that cannot be on a shortest solution path
Factor	factor a problem into independent subproblems
Apply Finite Differencing*	incrementally update the heuristic as a result of an operator application in the original problem
Precompute Lookup Table*	store the distance from goal to every reachable state
Collapse to Closed Form	collapse non-branching search to closed form formula of shortest path length

* = new: not in Absolver I.

x to location y, can be specialized to Move(1,x,y), which moves only tile 1 (variables are in lower case), by binding parameter y to 1. If the original operator is removed, all operators (i.e., all instances of Move(1,x,y)) will now only move tile 1. In effect, the operators that move all other tiles have been removed.

The set of relevant operators is actually constructed by backchaining on every literal in the goal statement for a given problem. For example, after applying drop(Blank), if the goal statement only specifies the location of tile 1, the relevant set of operators would contain only Move(1,x,y) (only the most general versions of each operator are kept in the set). Notice that if an operator can apply to some state on a shortest path to a goal state, then no operators more specific than it will be in the set of relevant operators for that goal. Therefore, all those operators that are irrelevant can be removed without increasing the length of the shortest path. While it may not be possible to precisely to compute the set of operators that will *definitely* apply to a state on a shortest path for a given goal statement without actually solving the original search problem, it is entirely possible to compute the set of those operators that *might* apply to a state on such a shortest path. That set is simply the set of relevant operators.

The Factor speedup partitions the literals in the goal statement and corresponding sets of relevant operators into sets of mutually *independent* goal literals and corresponding operators. Two sets of goal literals and corresponding operators are independent from each other iff no literal in the add set of an operator from one set unifies with a literal in the precondition set of an operator or a goal literal in the other set. Figure 3 shows how the set of independently solvable subproblems is used to compute the Manhattan Distance heuristic for the Eight Puzzle.

After applying drop(Blank) to the original problem, the resulting problem is factored into the set of independently solvable subproblems shown above the original problem. Each

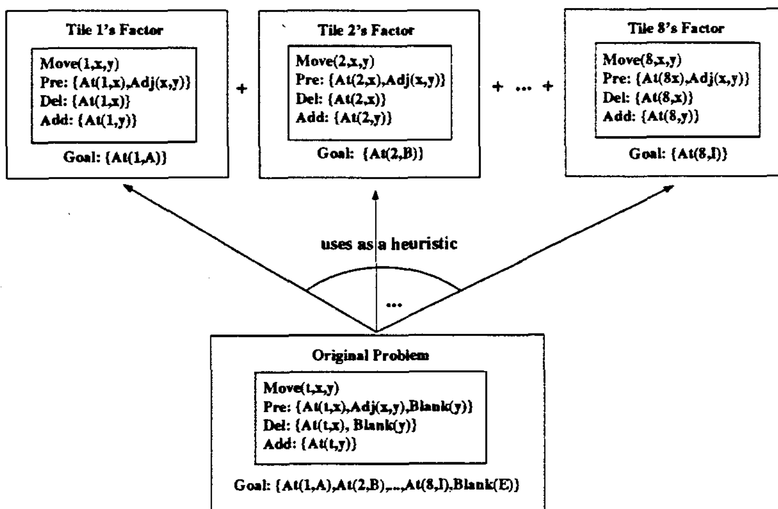

Figure 3. Independently solvable subproblems used to compute Manhattan Distance.

subproblem contains a set of goal literals and corresponding relevant operators. For example, the upper-leftmost subproblem contains goal literal set `{At(1,A)}`, which says tile `1` must be at location `A`, and the corresponding relevant operator set is `{Move(1,x,y)}`. An operator in this subproblem moves only tile `1` to adjacent locations; the least number of moves is the rectilinear distance to the goal location (`A`) of the tile. Since each of the subproblems is guaranteed to be independent from all the other subproblems because irrelevant operators have been removed, each can be solved independently from the others. For example, the moves in the upper-leftmost subproblem do not affect tiles 2–8. As a result, factoring into independent subproblems preserves admissibility. The sum of the lengths of the rectilinear distance to goal (i.e., the shortest path) for each of these subproblems is the Manhattan Distance.

In general, factoring reduces the complexity of search from the product to the sum of the size of the search spaces (number of states) for each independent subproblem. Given k independently solvable subproblems, each of which has branching factor b and depth d, the complexity of solving them together (i.e., without factoring) is $O((kb)^{kd})$, since operators for solving any one of the k subproblems apply at any point and the total depth of solving all k subproblems must be k times the depth of solving any one subproblem. In contrast, the complexity of solving the set of problems independently is $O(kb^d)$. For example, after applying `drop(Blank)` in an $n \times n$ sliding block puzzle, the resulting problem can be factored into a set of $n^2 - 1$ independently solvable problems, each of which has search space of size $O(n^2)$. In contrast, the size of the search space is $O((n^2)^{n^2-1})$ without factoring, since each of the $n^2 - 1$ tiles can be in any of the n^2 positions.

During actual problem-solving, the value returned by a heuristic generated from an abstracted factored problem can be incrementally updated a result of each move in the original problem. After computing the shortest path on the initial state for *every* factored subproblem, the shortest path for only *one* factored subproblem needs to be computed after each subsequent move in the original problem; the remaining subproblems do not need to be solved because each move in the original problem will only affect one abstracted subproblem. For example, if `Move(1,A,B)` is applied in the original problem, only the shortest path for the abstracted subproblem involving tile `1` needs to be updated to compute the Manhattan Distance for the resulting state. We call this idea *finite differencing*.[1] In general finite differencing speeds up search by roughly $O(k)$ with k independently solvable subproblems of the same complexity, since the value of the heuristic need be computed only once for all subproblems (i.e., initially) and subsequent computations involve only a single subproblem per move in the original problem. For example, the complexity of the Manhattan Distance heuristic is reduced from $O(n^4)$ to $O(n^2)$ by applying finite differencing. This complexity analysis ignores an initial $O(n^4)$ computation, which is amortized over all subsequent states of the problem instance.

Finally, given a hashing function and an efficient lookup algorithm, the distance from the abstracted goal to each abstracted state can be precomputed and looked up in constant time. Of course, it is possible to precompute a lookup table for the original problem, but the table size will be as large as the original search space, which is typically an exponential of the problem size. And if the original search space size is exponential, then it will take exponential time to fill the table. The objective is to precompute a lookup table when table size is a polynomial of the problem size or to precompute up to a prespecified limit

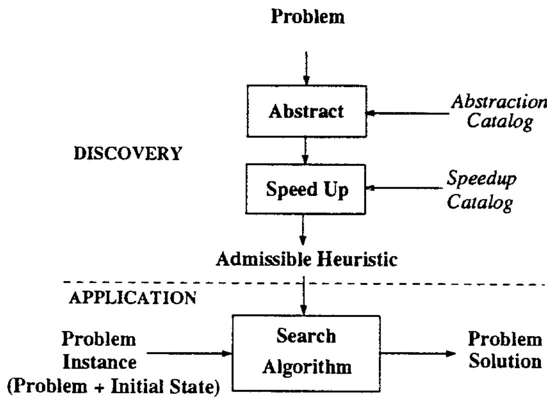

Figure 4. Our model for discovery and application of admissible heuristics.

when the number of states is exponential. Unlike all the other speedup transformations, which are automated, the decision to precompute a lookup table remains manual, since it depends on a good hashing scheme and an efficient lookup algorithm.

Figure 4 summarizes our model for discovering and applying 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). The basic idea is that abstractions coupled with speedups can reduce the complexity of computing the resulting heuristics. In particular, the resulting complexity reduction for $n \times n$ sliding block puzzles is as follows (complexity—number of states—of the original problem is leftmost):

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

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. Discovering effective heuristics amounts to finding those abstractions that can be sped up, since speedups might only be enabled after abstraction. The next section describes how Absolver II solves this problem.

4. Absolver II

Although the effort of discovering heuristics is amortized over all instances of a problem, blindly generating and testing each abstraction for speedup applicability is generally too expensive. For example, the number of combinations of `drop` abstractions with n relation names is 2^n. A good search control strategy should find effective heuristics quickly (if they exist) in this space.

Absolver II's means-ends search strategy relies on a "difference" table (table 3) of plausible abstracting transformations for each implemented speedup to identify and eliminate obstacles

Table 3. Plausible abstractions for each speedup transformation.

Abstraction	Speedup transformation			
	Factor	Remove redundant	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		

to applying speedups. 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; the lack of an "x" means that the abstracting transformation is not likely to lead to satisfying the speedup, given our experience. Finite Differencing is not shown because it is always applied after Factoring; Precompute is not shown because it is not automated (for the reasons mentioned in section 3).

Using this table, Absolver II halts and outputs the first heuristic it finds *subject to the following constraints* implicit in its search mechanism (paraphrased in everyday English):

1. The less information dropped to derive the heuristic the better (e.g., the less preconditions dropped the better).
2. All abstracted problems must be sped up.

Absolver II's search control strategy is actually comprised of three subprograms: Composer, Dropper, and Summarizer. The objective of Composer is to find an abstracted problem in which redundant operators can be removed by applying composition abstractions such as sum, thereby reducing the branching factor. The objective of Dropper is to find an abstracted problem that can be factored into at least two independent subproblems by applying information dropping abstractions such as drop_pre (irrelevant operators are removed prior to factoring). The objective of Summarizer is to find an abstracted problem in which redundant operators can be removed by applying mapping abstractions such as count.

Using the difference table, Absolver II's overall search strategy is to apply abstractions until it finds a problem that can be sped up. It first calls Composer. If Composer fails to find a problem in which redundant operators can be removed, then it calls Dropper. If Dropper cannot factor the problem into two or more independent subproblems, then Absolver II calls Summarizer as a last resort.[2] After Absolver II succeeds in finding an abstracted problem that can be sped up, it calls itself recursively on the resulting problem, thus resulting in a hierarchy of abstracted problems, which generates a hierarchy of heuristics. For example, if Dropper succeeds in finding a factorable set of subproblems, it factors the problem into the subproblems and then calls itself recursively on each of the subproblems. Each heuristic in the hierarchy 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 simple: it simply applies `count` (to every predicate) in attempt to find an abstracted problem in which redundant operators can be removed and, failing that, it applies `parity` (to every integer) for another attempt at removing redundant operators.

4.1. Composer: The search for redundant operators

Composer searches through the space of composition abstractions for those abstractions that lead to the removal of redundant operators. Instead of considering all possible such compositions, it considers only pairwise compositions and searches this space using a standard hill-climbing algorithm and a meta-heuristic in the form of a "similarity" coefficient. This meta-heuristic returns the number of operator pairs in which the candidate composition of the two array elements is equal; the larger the similarity coefficient, the more likely that a particular composition will lead to a redundant operator (i.e., making one operator identical to another by applying compositions). Composer applies the similarity coefficient to each pairwise candidate composition and then proceeds in the direction of that pairwise composition with the largest similarity coefficient; ties are broken arbitrarily.

When the number of uninstantiated operators after redundant operators are removed is 75% of the number of original operators. Absolver II is called recursively to build a hierarchy of heuristics. (In effect, Composer calls itself recursively because Absolver II typically ends up called Composer again.) The 75% value, which we chose initially and have not had to change, is a rough indicator that the branching factor of the abstracted 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, the object of which is to orient each of the concentric disks such that the numbers on each radius (labeled $R1$–$R8$ in figure 5a) sum to 12, Composer generates the hierarchy of abstracted problems shown in figure 5b–d. Each problem in this hierarchy generates an admissible heuristic for the preceding level.

The Diameters problem, shown in figure 5b, is an abstraction of the original Fool's Disk problem where the following pairs of radii numbers are added using the `sum` transformation: $R1$ and $R5$, $R2$ and $R6$, $R3$ and $R7$, and $R4$ and $R8$. For example, the figure shows how the outer disk's numbers for radius $R1$ and $R5$ are summed to form the outer disk's number of a new radius called $R1R5$. The Perpendicular Diameters problem, shown in figure 5c, is an abstraction of the Diameters problem where perpendicular diameters are summed. For example, in the Perpendicular Diameters problem, the composite $R1$ and $R5$ is summed with the composite $R3$ and $R7$. The All Numbers problem, shown in figure 5d, is an abstraction of the Perpendicular Diameters 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 ∞ for a state at any level of abstraction (i.e., an exhaustive search failed), then the state can be pruned. In effect, the heuristics are actually used as a solvability test rather than a distance estimate: if the abstracted goal cannot be reached in the abstract space, then the original goal cannot be reached in the original space. In problems where

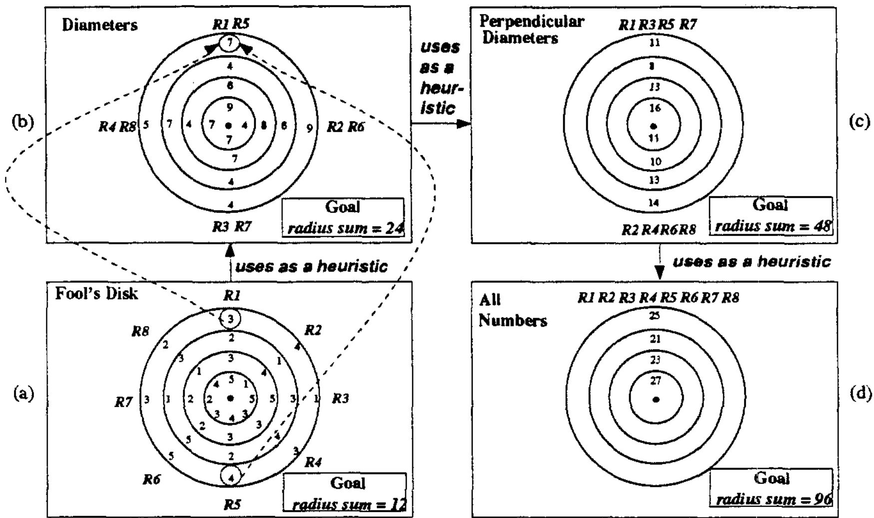

Figure 5. The hierarchy of heuristics discovered by Absolver II (circled radius values show example composition).

all solutions have the same length (e.g., the length is 4 for the Fool's Disk), the heuristics Absolver II discovers act as solvability tests rather than distance estimators. 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. The next few paragraphs describe how Composer finds these heuristics.

The original Fool's Disk problem is represented by states with an integer array of length 8; each element in this array represents the current radius sum and operators that place each disk in one of eight orientations. For example, the initial state is represented by the array ⟨0, 0, 0, 0, 0, 0, 0, 0⟩, and an operator that places the outermost disk in the orientation shown in figure 5a would have as its integer delta constant array ⟨3, 4, 1, 3, 4, 5, 3, 2⟩. Since there are four disks and eight orientations per disk, there are 32 such operators, each representing the placement of a particular disk in a particular orientation. Using these 32 operators, Composer computes the values for the similarity coefficient on each candidate combination of radius pairs. The results are shown in table 4. Each entry in the table is the similarity coefficient on the corresponding row and column. For example, the table shows that the sum of $R1$ and $R2$ is equal in 68 pairs of operators. Moreover, composing either of $R1$ with $R5$, $R2$ with $R6$, $R3$ with $R7$, or $R4$ with $R8$ is more likely to lead to operators becoming identical than composing another pair of radii.

Composer *arbitrarily* picks one of these highest-ranking candidate compositions and applies it. In our example, it picks the composition of $R1$ and $R5$. After applying this composition, it finds that there are still 32 operators in the resulting problem, so it again computes the similarity coefficient (this time including a new radius called $R1R4$). Again, the highest ranking compositions are $R2$ with $R6$, $R3$ with $R7$, and $R4$ with $R8$. Composer arbitrarily picks one of these ($R4$ with $R8$) and applies it. Even after this combination there are still

Table 4. Similarity coefficients for composing each radius pair (most promising compositions are highlighted).

	R2	***R3***	***R4***	***R5***	***R6***	***R7***	***R8***
R1	68	70	81	112	81	70	68
	R2	68	70	81	112	81	70
		R3	68	70	81	112	81
			R4	68	70	81	112
				R5	68	70	81
					R6	68	70
						R7	68

Table 5. Similarity coefficients for composing each diameter pair (most promising compositions are highlighted).

	R2R6	***R3R7***	***R4R8***
R1R5	14	16	14
	R2R6	14	16
		R3R7	14

32 operators, so Composer continues. Finally, after combining $R3$ with $R7$ and then $R2$ with $R6$, the number of operators shrinks to 16 (well within 75% of the original 32), which results in the Diameters problem.

Next, Composer calls itself recursively on the Diameters problem to obtain a heuristic that can be used for solving the Diameters problem. Table 5 shows the results of applying the similarity coefficient to the candidate compositions. The compositions of $R1R5$ with $R3R7$ and $R2R6$ with $R4R8$ top the list; Composer arbitrarily chooses the first one. After applying this composition, the number of operators shrinks to 15 because one redundant operator is removed. Since this number is still not within 75% of 16, Composer continues. After composing $R1R5$ with $R3R7$, the size of the operator set is reduced to 8, which results in the Perpendicular Diameters problem. Finally, Composer calls itself recursively to produce the All Numbers problem, where $R1R3R5R7$ and $R2R4R6R8$ *are composed.*

The complexity of each non-recursive call to Composer is dominated by the complexity of computing the similarity coefficient, whose complexity is $O(m^2n^2)$ for m operators and a size n array, since each pair of operators must be examined and the similarity coefficient will be applied to each pair of array elements. Since Composer will call itself recursively at most $O(n)$ times, the total complexity of it is $O(m^2n^3)$.

4.2. Dropper: The search for factorability

Dropper consists of two subprograms: Pairwise and Combine. For each pair of literals g_1 and g_2 in the set of goal literals and set of operators for a problem, Pairwise drops the following items (in order) until the pair of literals can be achieved independently from each other:

1. *Preconditions* that cause *direct* interaction: for every operator o_1 that can directly add g_1 (i.e., the operator contains an element in its *add* set that unifies with g_1), and every operator o_2 that can directly add g_2, drop those preconditions of o_1 that unify with an element of o_2's add set and vice versa.
2. *Other goal literals* that must be dropped in order to make the pair of goal literals independent: for every operator o_1 that can directly add g_1, and every operator o_2 that can directly add g_2, drop those goal literals that can be on the intersection of the add sets of operator o_1 and o_2. This second step is required because a third goal literal may prevent factoring the pair of literals.
3. *Other preconditions* that cause *indirect* interaction: if an operator that can lead to one achieving one literal adds a precondition of an operator that can lead to the other achieving the other literal, then drop the precondition.

Computing these drop sets for every pair of goal literals is important because these drop sets can then be combined to enable factoring of the entire set of goal literals and corresponding abstracted relevant operators. The abstraction `drops` is not explicitly applied because it is defined in terms `drop_pre` and `drop_goal` which are applied.

For example, given the two literals `At(1,A)` and `At(2,B)`, figure 6 shows that `drop_pre(Blank,Move)` will make the pair of literals independent from each other (step 1). Operators of the form `Move(1,x,A)` move tile 1 from location `x` to location `A` and directly add `At(1,A)`, and operators of the form `Move(2,x,B)` directly add `At(2,B)`. Since `Move(1,x,A)` places the blank in some location (possibly location `B`) and `Move(2,x,B)` places the blank in some location (possibly location `A`), it is possible for each operator to add a precondition required by the other. Dropping `Blank` from the precondition of the `Move` operator makes the application of one operator independent of the results obtained by the other. As a result, Pairwise applies `drop_pre(Blank,Move)`.

In step 2, since `Move(1,x,A)` and `Move(2,x,B)` place `Blank` in some location, `Blank` must additionally be dropped from the goal in order to make the literals `At(1,A)` and `At(2,B)` independent from one another. If this were not dropped, the pair of literals—though independent from each other—cannot be independently achieved from positioning the `Blank`. That is, one achieved goal literal may have to be undone to position the `Blank`.

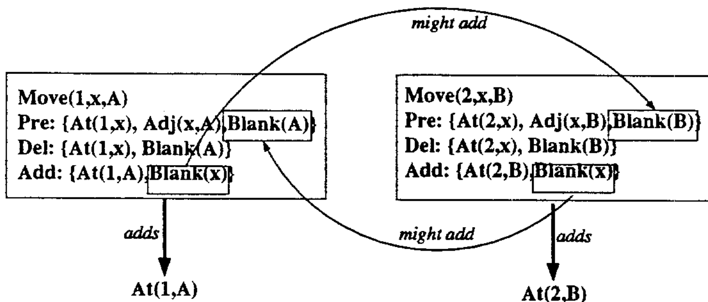

Figure 6. Dropping the blank from preconditions enables factoring of the two goal literals.

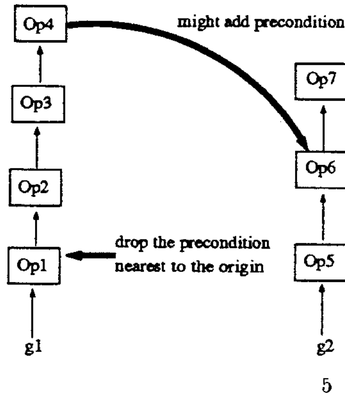

Figure 7. Locating the earliest cause of non-independence.

Consequently, Pairwise applies drop_goal(Blank) to enable factorability of the two goal literals.

In step 3, Pairwise constructs a chain of operators by backchaining from each literal and each precondition of operators already on the chain. Although each precondition can have a separate chain, Pairwise constructs the chains depth-first until it finds a reason for non-independence. If no such reason is found, it backtracks to a precondition whose chain has not yet been constructed. If an operator is detected that can add a precondition required by an operator in the other chain, it drops the (single) precondition of the operator that is nearest to the origin of the chain, even though the offending operator might be at a later point in the chain. Figure 7 shows this situation for a particular pair of literals and their associated chains. Once this precondition is dropped, a new chain is then constructed and the process repeats until the literals become independent. In our Eight Puzzle example, Pairwise actually succeeds after one such iteration because the two literals are already independent with respect to dropping Blank from operator preconditions.

Since Pairwise can *overestimate* the number of preconditions that must be dropped to ensure factorability, it calls a postprocessor that attempts to reduce the number of dropped preconditions by greedily adding back dropped preconditions and testing if independence is still preserved (i.e., pass through the previous three steps without dropping any preconditions or literals).

Even after Pairwise has computed the set of preconditions and goal literals to drop for each pair of goal literals, the entire set of goal literals and corresponding abstracted relevant operators may not be factorable because of interactions among goal literals triples, quadruples, and higher combinations. In short, the drop sets may have to be combined to make the entire set of goal literals and corresponding abstracted relevant operators factorable. Combine does just that. It unions the drop sets produced by Pairwise by using the depth first iterative deepening search algorithm (Korf, 1985) with two meta-heuristics to further reduce search (iteration is on the number of precondition and goal literal drops):

1. Union those drop sets first that make the most literal pairs independent—this action might be more likely to lead to faster convergence to factorability while reducing the number of drops.

2. Ignore those drop sets that drop a goal literal only to make *one* independent from another. Such drop sets are less likely to lead to factorability, since they do not affect the factorability of other goal literals.

This search algorithm tends to minimize the number of preconditions and goal literals to drop and hence tries to abstract as little as possible. 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 of the iterative deepening algorithm. Combine is then called recursively on each of the factored subproblems. Finally, it terminates successfully, since no other factorable subproblems can be obtained within any of the factored subproblems. The resulting set of subproblems is the AND tree shown in figure 3, which is used to compute the Manhattan Distance heuristic. Each recursive call produces another level of the tree.

The complexity of Dropper is dominated by the third step of Pairwise, which is also called for additional independence tests for greedily adding back dropped preconditions as described above and 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 operators instances—operators whose parameters are bound to constants—that add 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 goal literal). That is, the number of preconditions that Pairwise examines is porportional to the number of paths from a goal literal to each operator in the chain.

5. Experimental results

Table 6 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).[3] 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 Absolver II's catalog of abstracting transformations and multiplying by 100. The space size is a conservative estimate in that it includes only those abstractions that Absolver II actually considers in its search. For example, the space size for deriving the Manhattan Distance is 4096, since Absolver II drops operator preconditions and literals from the goal to derive the heuristic, and there are 9 goal literals and 3 preconditions (2^{9+3} = 4096). Since arrays are not part of our problem specification for the Eight Puzzle, Absolver II does not apply composition abstractions. In contrast, to derive the Fool's Disk heuristic, Absolver II only applies sum (to the 8 radii) and not other abstractions; 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 (we define "effective" as roughly one or more order of magnitude of speedup over exhaustive search). The novel effective heuristics include the Center-Corner, X-Y, Box Distance, and the Nearly Opposite Sides heuristic.

Table 6. Admissible heuristics discovered by Absolver II.

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}

* = novel.
\+ = effective.

The *Center-Corner* heuristic computes the minimum number of moves required to get just the center cubies in place plus the minimum number of moves required to get just the corner cubies in place for the $3 \times 3 \times 3$ Rubik's Cube. This heuristic resulted in roughly eight orders of magnitude speedup with *IDA** over the expected time for breadth-first (exhaustive) search for long solutions. This result makes it the first known (non-trivial) admissible heuristic for the Cube. For search problems such as the Rubik's Cube, we chose the *IDA** algorithm to evaluate the admissible heuristics discovered by Absolver II because this algorithm is standard for evaluating admissible heuristics. Moreover, its results can often be analytically compared to those of breadth-first search (i.e., by computing the average branching factor and depth, the total number of states expanded by breadth-first search and solution generation time can be estimated). To make the heuristic more effective, the Corner portion was precomputed up to depth 6 from the goal and the Center portion was entirely precomputed for these experiments.

The *X-Y* heuristic computes the minimum number of column-adjacent blank swaps to get all tiles in their destination column plus the minimum number of row-adjacent blank swaps to get all tiles in their destination row for sliding block puzzles. The heuristic turned out to be the best admissible heuristic for the Eight Puzzle with *IDA**, expanding 1.8 times fewer states than the Linear Conflict heuristic (Hansson et al., 1992), an adjusted, more accurate version of the Manhattan Distance heuristic. The version of the X-Y heuristic that we used was entirely precomputed: the X-Y. Without precomputation, the heuristic is less effective than the Linear Conflict heuristic, but better than Gaschnig's n-Maxswap heuristic (Gaschnig, 1979) and better than no heuristic at all (breadth-first search). The Center-Corner and X-Y heuristics demonstrate that abstraction coupled with precomputation can produce effective heuristics.

The *Box Distance* heuristic computes the minimum number of rooms each box must pass through to reach its destination for the Rooms World. It expanded 267 times fewer states using *IDA** than breadth-first search for the Rooms World problem, shown in figure 8.

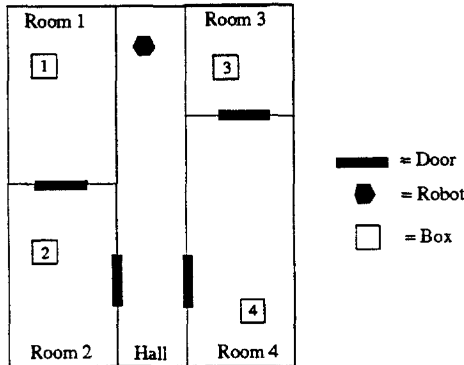

Figure 8. A state in the Rooms World problem.

The *Diameters* heuristic is the Fool's Disk heuristic described in section 4. It expanded 45.41 times fewer states than exhaustive search. Both were evaluated using standard depth-first search with backtracking rather than with IDA^* because all solutions have the same length (in this case, 4). As described in the previous section, the heuristic acts as a solvability test during this search. The Diameters heuristic relies on the hierarchy of heuristics described in section 4 for faster computation.

The *Nearly Opposite Sides* heuristic is an Instant Insanity analogue of the Diameter's heuristic in that opposite side colors are composed. (In Instant Insanity the objective is to build a stack of four cubes with variously colored faces such that no stack side contains two or more faces of the same color.) It expanded 2.61 times fewer states than exhaustive search with the same depth-first backtracking program as for the Fool's Disk. The Nearly Opposite Sides heuristic relies on a hierarchy of heuristics analogous to those of the Diameters heuristic, which Absolver II also discovered. The X-Y, Box Distance, Diameters, and Nearly Opposite Sides heuristics demonstrate that heuristics can be effective even though they are computed by search.

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 $n \times n$ puzzles because the derived heuristic uses search to compute the minimum number of moves needed to get each tile from its current location to its goal location. This analysis assumes finite differencing and ignores a one-time $O(n^4)$ initial computation, which is amortized over all subsequent Manhattan Distance computations

Absolver II derived several inferior heuristics. In the 2-D Routing domain, the object of which is to find a shortest rectilinear routing from each source point to multiple target destinations for multiple signals, it derived the *Unvisited Signals* heuristic, which counts of the number of non-reached signal locations. The reason it derived this heuristic, instead of the more accurate Steiner Tree heuristic, which returns the length of a minimum rectilinear spanning tree and which we derived by hand (Prieditis, 1990), is 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 accurate 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 Uniprocessor Scheduling problems for the same reason we failed to find one by hand (Prieditis, 1990): all abstractions that could be sped up removed too many important details and therefore resulted in relatively inaccurate heuristics. For example, to obtain the factorable abstracted problem that results in the *Unassigned Jobs* heuristic for the Uniprocessor Scheduling problem, the time and seriality constraints must be dropped. The resulting heuristic, which returns the minimum costs of unassigned jobs, is relatively inaccurate. 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 affect many parts of the goal statement or make many operators subsequently inapplicable in the original problem), and all abstracted problems are characterized by low goal or operator "interference" (few operators directly affect many parts of the goal statement or make many operators subsequently inapplicable in the original problem). Of course, it may be that to derive effective admissible heuristics for such domains requires abstractions and speedups beyond our model or simply that we were not able to find a "good" formulation in which to derive heuristics.

6. Relation to other work in machine discovery and learning

Absolver II differs from traditional data-driven scientific discovery systems, such as Bacon (Langley et al., 1987), Coper (Kokar, 1986), Fahrenheit (Żytkow et al., 1990), and others (Schaffer, 1990), in that it does not rely on induction from data for its discovery process. Instead, because its discovery process is driven by the goal of obtaining an abstracted problem that can be sped up, it can use techniques such as means-ends analysis to focus on its discovery process.

Absolver II is similar to discovery systems such as AM (Lenat, 1977), which discovers mathematical concepts, and Eurisko (Lenat, 1982, 1983a, 1983b), which discovery heuristics for AM-like discovery systems. Both systems discover concepts by searching the space of syntactic transformations of existing concepts. AM appears to work well precisely because such transformations are likely to lead to other mathematically interesting concepts (Lenat & Brown, 1984). Similarly, because the class of admissible heuristics that Absolver II discovers are ultimately defined in terms of syntactic features of a problem, syntactic transformations of a problem are likely to lead to interesting heuristics. In short, AM and Absolver II seem to be searching a space rich in the type of objects they are trying to discover. Had they instead searched the space of binary representations of a problem by applying bit inversions, they would have no doubt failed miserably, because the relationship between the manipulation of one particular bit and the original problem is at best tenuous. Indeed, Eurisko was not as successful as AM precisely because it lacked a natural form/content mapping between its representation and the meaning of its concepts.

Absolver II differs from systems that *learn* heuristics in that it does not rely on generalization from example solutions or on an external teacher who supplies example problems or solutions. Some of these learning systems rely on simplying assumptions to generate heuristics. For example, POLLYANNA (Ellman, 1988) generates heuristics for the card game Hearts by applying generic probabilistic simplifying assumptions to an intractable domain theory; the resulting heuristics are then empirically tested. Similarly, MetaLEX (Keller, 1987) generates heuristics for search tasks by using performance statistics to guide its simplification of an intractable search domain theory. The simplifications differ from abstracting transformations in that they do not ensure admissibility.

Other systems that generate non-admissible heuristics use induction over example states on solution paths. For example, induction over states on solution paths can be used to learn the class of states for which it is useful to apply operators along such paths (Mitchell et al., 1983; Langley, 1983). These heuristics are not admissible and evaluate the usefulness of an operator rather than estimate distance to goal.

As with induction, examples are used to derive heuristics in several other learning systems. One system adjusts the coefficients of a polynomial-based state evaluation function in response to positive and negative outcomes in the game of checkers (Samuel, 1963). Another derives heuristics by clustering information obtained from searches (Rendell, 1976). Yet another learns a state evaluation function by linear regression over a set of states with numerical features (Christensen & Korf, 1986). In contrast, explanation-based learning systems analyze solutions or non-solutions to learn heuristics for when to apply or not apply an operator (Minton, 1988; Mostow & Bhatnagar, 1987; Bhatnagar & Mostow, 1990). Again, these approaches result in non-admissible heuristics

Absolver II (the Composer subprogram in particular) is similar to INFIN (Oyen, 1975), which discovers *primitive* problem-solving invariants, and DGBS (Ernst & Goldstein, 1982), which discovers *higher-level* invariants that are then converted to problem-solving strategies for the General Problem-Solver (GPS) program. All three programs apply composition abstractions, though for different reasons. For every operator, INFIN computes those items in a state (a state is represented as a set of integer or bag arrays) that are invariant—unchanged as a result of applying the operator. It then removes these invariants from the state and tries to find pairs of items that are invariant. This cycle continues with triples and higher combinations of items until all invariants have been removed. The resulting set of invariants are called *primitive*. DGBS then takes these primitive invariants and tries to combine them using addition, multiplication, and bag union to obtain problem-solving subgoals that can be serially processed. For example, INFIN coupled with DGBS discovered the following three-step strategy for the Fool's Disk:

1. Using the original set of 32 Fool's Disk operators, find a sequence of operators leading from a given initial state i to a state g_1, where the sum of numbers on the horizontal and vertical diameters is 48.
2. Using only those operators that preserve the horizontal and vertical diameter sum of 48, find a sequence of operators leading from g_1 to a state g_2, where each diameter sums to 24.
3. Using only those operators that preserrve the diameter sum of 24, find a sequence of operators leading from g_2 to the original goal states.

Of course, backtracking between steps might be needed because the states g_1 or g_2 might not be on the solution path. In essence, this strategy attempts to make problem-solving more efficient by setting up subgoals and reducing the branching factor between subgoals by only applying operators that preserve previous subgoals (Guvenir & Ernst, 1990).

All three programs are sensitive to problem formulation.[4] For example, strong conditions on the problem formulation are often required to discover GPS-like strategies: if the Fool's Disk is formulated with 45° rotations, which is equivalent to the original formulation, DGBS discovers no meaningful invariants. Absolver II is somewhat less sensitive to problem formulation because heuristics may still be discoverable when efficient problem-solving strategies cannot.

INFIN and DGBS might benefit from the discovery strategies and transformations used by Absolver II and vice versa. For example, INFIN might be able to discover more complex primitive invariants by first applying other abstracting transformations. Conversely, Absolver II might be able to discover efficient solvability tests based on heuristic resulting from combining primitive invariants as in DGBS.

7. Conclusions, shortcomings, and future work

Absolver II tractably discovered many known admissible heuristics and some novel effective admissible heuristics. Some of these heuristics are still effective even though they are computed by search; others used precomputation to make them more effective. Search-computed heuristics might be important in domains such as the Room's World, for which effective closed-form heuristics might not exist.

Since 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 accurateness of certain admissible heuristics by taking into account interactions in the base level between independently solvable factored subproblems in the abstract level. For example, the Linear Conflict heuristic might be derivable by incrementing the Manhattan Distance heuristic by at least 1 for each interaction found in the original problem between independent factors with only one shortest path solution. We have been able to boost by hand the accurateness of a popular project scheduling heuristic using a similar technique called *reconstitution* (Janakiraman & Prieditis, 1992), which efficiently adds back previously abstracted information while maintaining admissibility. Ultimately, we would like to automate this process.

Finally, Absolver II sometimes derives inferior heuristics because it applies too many abstracting transformations to obtain a problem that can be sped up. Absolver II could assume, for example, that a problem is factorable until experience during problem-solving proves otherwise. Once a problem is proved to be non-factorable, the reason could be located and eliminated by abstraction. Of course, since the original assumption of factorability

ignores the test for independence, the resulting heuristics would not be guaranteed to be admissible. Sacrificing admissibility for ease of discovery and ease of computation may be the only method of dealing with the expense of discovering and computing certain heuristics.

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

Acknowledgments

Thanks go to Jack Mostow, Tom Mitchell, Alex Borgida, Saul Amarel, Haym Hirsh, Christina Chang, Mukesh Dalal, Sridhar Mahadevan, and Prasad Tadepalli for many discussions and helpful advice on this research. Thanks also go to Rich Cooperman, for testing the Rubik's Cube heuristics; Rich Korf, for supplying the the *IDA** program used to collect data for the Manhattan Distance heuristic; and Jan Żytkow and two anonymous referees, for helping to improve the quality of this article. The work described here was supported in part by the National Science Foundation (NSF) under grant numbers IRI-9109796 and DMC-86610507, and by the Defense Advanced Research Projects Agency (DARPA) under grant number N00014-85-K-0116.

Notes

1. Finite differencing was originally introduced in the context of programming languages (Paige & Koenig, 1982; Moshi, 1989): differentiate over a computable function and then use that differential to incrementally compute the function. In our case, the differential requires search to be computed.
2. Absolver II collapses problems to closed form whenever possible: it does not explicitly search for such problems.
3. The domains, problem formulations, methods for choosing good formulations, derivations, and performance of the resulting heuristics are detailed by Prieditis (1990).
4. Finding problem formulations for efficient problem-solving is a difficult problem (Amarel, 1968).

References

Amarel, S. (1968). On the representation of problems of reasoning about actions. In *Machine intelligence* (Vol. 3). Edinburgh, Scotland: Edinburgh University Press.

Bhatnagar, N., & Mostow, J. (1990). Adaptaive search by explanation-based learning of heuristic censors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*. Boston, MA: American Association for Artificial Intelligence.

Chenoweth, S., & Davis, H. (1991). High-performance a* search with rapidly growing heuristics. In *Proceedings IJCAI-12*. Sydney, Australia: International Joint Conferences on Artificial Intelligence.

Christensen, J., & Korf, R. (1986). A unified theory of heuristic evaluation functions and its applications to learning. In *Proceedings AAAI-86* (pp. 148–152). Philadelphia, PA: American Association for Artificial Intelligence.

Ellman, T. (1988). Approximate theory formation: an explanation-based approach. In *AAAI88* (pp. 570–574). St. Paul, MN: American Association for Artificial Intelligence.

Ernst, G., & Goldstein, M. (1982). Mechanical discovery of classes of problem-solving strategies. *JACM, 29*(1), 1–23.

Fikes, R., Hart, P., & Nilsson, N.J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3*(4), 251–288. Also in B.L. Webber & N.J. Nilsson (Eds.), *Readings in Artificial Intelligence*. Palo Alto, CA: Tioga.

Gaschnig, J. (1979). A problem-similarity approach to devising heuristics. In *Proceedings IJCAI-6* (pp. 301–307). Tokyo, Japan: International Joint Conferences of Artificial Intelligence.

Guida, G., & Somalvico, M. (1979). A method for computing heuristics in problem solving. *Information Sciences*, *19*, 251–259.

Guvenir, H., & Ernst, G.W. (1990). Learning problem solving strategies using refinement and macro generation. *Artificial Intelligence*, *44*(1–2), 209–243.

Hansson, O., Mayer, A., & Yung, M. (in press). Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*.

Held, M., & Karp, R. (1970). The traveling salesman problem and minimum spanning trees. *Operations Research*, *18*, 1138–1162.

Janakiraman, B., & Prieditis, A. (1992). Generating effective admissible project scheduling heuristics by abstraction and reconstitution. In *Proceedings of the AAAI 1992 Spring Symposium Series: Practical Approaches to Planning and Scheduling*. Stanford, CA: American Association for Artificial Intelligence.

Keller, R. (1987). *The role of explicit contextual knowledge in learning concepts to improve performance*. Ph.D. thesis, Rutgers University, New Brunswick, NJ.

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

Kokar, M. (1986). Discovering functional formulas through changing representation. In *Proceedings AAAI-86*. Philadelphia, PA: American Association for Artificial Intelligence.

Korf, R. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(2), 97–109.

Langley, P. (1983). Learning effective search heuristics. In *Proceedings IJCAI-8*. Karlsruhe, West Germany: International Joint Conferences on Artificial Intelligence.

Langley, P., Simon, H.A., Bradshaw, G., & Żytkow, J. (1987). *Scientific discovery: Computational explorations of the creative process*. Cambridge, MA: MIT Press.

Lawler, E., & Lenstra, L. (1984). *The Traveling Salesman problem*. New York: John Wiley and Sons.

Lenat, D.B. (1977). On automated scientific theory formation: A case study using the AM program. In *Machine intelligence* (Vol. 9). New York: Halsted Press.

Lenat, D.B. (1982). The nature of heuristics. *Artificial Intelligence*, *19*, 189–249.

Lenat, D.B. (1983a). The nature of heuristics ii: Background and examples. *Artificial Intelligence*, *21*, 31–59.

Lenat, D.B. (1983b). The role of heuristics in learning by discovery: Three case studies. In *Machine learning*, Los Altos, CA: Morgan Kaufmann.

Lenat, D.B., & Brown, J.S. (1984). Why AM and EURISKO appear to work. *Artificial Intelligence*, *23*, 269–294.

Minton, S. (1988). *Learning effective search-control knowledge: An explanation-based approach*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Mitchell, T.M., Utgoff, P.E., & Banerji, R.B. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine learning: An artificial intelligence approach*. Palo Alto, CA: Tioga.

Moshi, A. (1989). *Transformations for backtracking SETL programs*. Ph.D. thesis, New York University.

Mostow, J., & Bhatnagar, N. (1987). Failsafe—a floor planner that uses EBG to learn from its failures. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI87)*, Vol. 1 (pp. 249–255). Milan, Italy: Morgan Kaufmann.

Mostow, J. & Prieditis, A. (1989). Discovering admissible heuristics by abstracting and optimizing. In *Proceedings IJCAI-11*, Detroit, MI: International Joint Conferences on Artificial Intelligence.

Nilsson, N.J. (1980). *Principles of artificial intelligence*. Palo Alto, CA: Morgan Kaufmann.

Oyen, R.A. (1975). *Mechanical discovery of invariances for problem solving* (Technical Report 1168). Cleveland, OH: Computer Engineering Department, Case Western University.

Paige, R., & Koenig, S. (1982). Finite differencing of computable expressions. *ACM TOPLAS*, *4*(3), 402–454.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem-solving*. Reading, MA: Addison-Wesley.

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

Prieditis, A. (1990). *Discovering effective admissible heuristics by abstraction and speedup: A transformational approach*. Ph.D. thesis, Rutgers University, New Brunswick, NJ.

Rendell, L. (1976). *A method for automatic generation of heuristics for state-space problems* (Technical Report CS-76-10). Waterloo, Ontario: University of Waterloo, CS Department.

Samuel, A.L. (1963). Some studies of machine learning using the game of checkers. In *Computers and Thought*. New York: McGraw-Hill.

Schaffer, C. (1990). A proven domain-independent scientific function-finding algorithm. In *Proceedings AAAI-90*. Boston, MA: American Association for Artificial Intelligence.

Valtorta, M. (1984). A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences, 34*, 47–59.

Żytkow, J., Zhu, J., & Hussam, A. (1990). Automated discovery in a chemistry laboratory. In *Proceedings AAAI-90*. Boston, MA: American Association for Artificial Intelligence.

Received November 26, 1990
Final Manuscript May 26, 1992