Discovering Admissible Heuristics by Abstracting and Optimizing: A Transformational Approach

Jack Mostow and Armand E. Prieditis*
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

We present an implemented model for discovering a class of state-space search heuristics. First, *abstractions* of a state-space problem are generated by dropping information from the problem definition. An optimal solution path for any such abstracted problem gives a lower bound on the true distance to the goal. This bound can be used as an admissible evaluation function for guiding the base-level search. Moreover, if the abstracted goal is unreachable from an abstracted state, the original state can safely be pruned. However, using exhaustive search to evaluate the abstracted problem is generally too slow. Therefore, *optimization* is used to speed up the computation of the lower bound (or solvability test), for example by factoring the abstracted problem into independent subproblems. We analyze the conditions under which the resulting heuristic is faster than brute force search. Our implementation, named ABSOLVER, has several general transformations for abstracting and simplifying state-space problems, including a novel method for problem factoring. ABSOLVER appears to be the first mechanical generator of heuristics guaranteed to find optimal solution paths. We have used it to derive known and novel heuristics for various state space problems, including Rubik's Cube.

*The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract number N00014-85-K-0116, in part by the National Science Foundation (NSF) under Grant Number DMC-8610507, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), an Advanced Technology Center of the New Jersey Commission on Science and Technology, at Rutgers University, Piscataway, New Jersey. The opinions expressed in this paper arc those of the authors and do not reflect any policies, either expressed or implied, of any granting agency. We thank Saul Amarel, Tony Bonner, Alex Borgida, Christina Chang, Jan Chomicki, William Cohen, Mukesh Dalai, Richard Korf, Michelle Kraus, Pat Langley, Sridhar Mahadevan, Tom Mitchell, Stan Raatz, Lou Steinberg, Chris Tong, and Bob Webber for their helpful comments.

1 Introduction

Finding optimal (least cost) solutions to large state-space problems is generally intractable without good admissible heuristics (evaluation functions that return lower-bound estimates of distance to goal). When coupled with search algorithms that ensure optimality, like *A** [Nilsson, 1980] or iterative-deepening *A** *(IDA*)* [Korf, 1985a], such heuristics can reorder the search so that solutions are found much earlier. They can also reduce search by pruning states that lie more than a specified distance from the goal. This distance may be the exact solution length for problems where it is known *a priori,* an upper bound on acceptable solution length, or just infinity, in which case the predicate tests whether the goal is reachable at all from a given state.

However, good admissible heuristics can be hard to find. For example, after extensive study, Korf was unable to find a single good heuristic evaluation function for Rubik's Cube [Korf, 1985b]. He concluded that "if there does exist a heuristic, its form is probably quite complex."

The long-term goal of our research is to develop a system that can discover good admissible heuristics automatically, or at least with less user effort than discovering them by hand. The main contribution of this paper is a model for discovering such heuristics, and its partial implementation in a system called ABSOLVER. We next describe how it works, and then evaluate it as an explanatory model, as a generative model, and as an automatic discovery engine.

2 ABSOLVER

Our approach, illustrated in Figure 1, derives admissible heuristics from abstractions of a state-space problem. ABSOLVER is initially given a STRIPS-like representation of a problem class. The distinction between a problem class and a problem instance is important because the effort of discovering a heuristic can be amortized over all instances of the problem class.

ABSOLVER generates abstractions by dropping information from this description via a series of abstracting transformations chosen from a catalog. An optimal solution path for any resulting abstracted problem gives a lower bound on true distance to goal. This lower bound can then be used as an admissible evaluation function

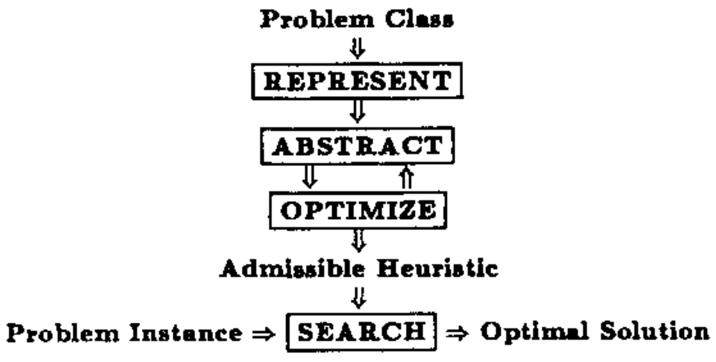

Figure 1: ABSOLVER's Model of Discovery

for guiding the base-level search. Moreover, if the abstracted goal is unreachable from an abstracted state (within the specified upper bound), the original state can safely be pruned.

However, using brute-force search to solve the abstracted problem is generally too expensive. Suppose that using the abstracted problem to estimate distance to goal reduces the branching factor of the base level search from b to c. Base level depth is unchanged, so c^d base level states must be expanded and evaluated. Without optimization, it costs $b_a^{d_a}$ to perform each evaluation by searching an abstracted space with branching factor b_a and depth d_a. This strategy pays off only if $c^d * b_a^{d_a} < b^d$. Therefore an optimization phase is needed to speed up the computation of the heuristic, for example by factoring the abstracted problem into independent subproblems.

Finally, since this model can generate good *and* bad heuristics, the effectiveness of a heuristic is evaluated on sample instances of the problem.

We now use a running example to illustrate each component of this model in more detail.

2.1 Representation

Our model starts with a STRIPS-style problem class representation consisting of a goal and a set of operators. To avoid anomalous abstractions, we exclude negation and universal quantifiers, and require the delete list to be a subset of the precondition list. An individual problem instance consists of finding a sequence of operator applications leading from a specified initial state to a state matching the goal.

For example, the Eight Puzzle can be represented in terms of the predicates $at(T,S)$ (tile T is on square S), $blank(S)$ (square S is blank), and $adj(S,S')$ (square S is adjacent to square S'). The goal is represented as a list of subgoals, e.g., $\{at(a,1), at(b,2), \ldots, \boxed{blank(6)}\}$, where tiles are lettered a through g and squares are numbered 1 through 9. A single operator suffices:

$move(T,S,S')$ — move T from square S to S'

pre: $at(T,S), \boxed{blank(S')}, adj(S,S')$

add: $at(T,S'), \boxed{blank(S)}$

del: $at(T,S), \boxed{blank(S')}$

The boxes will be explained later.

The heuristics generated by ABSOLVER depend to some extent on the initial representation. For example, ABSOLVER produces somewhat different results given a Cartesian representation in terms of the following predicates. The predicate $xloc(T,X)$ holds if tile T is at x-coordinate X; $yloc(T,Y)$ if tile T is at y-coordinate Y; $xlocb(X)$ if the blank is at x-coordinate X; $ylocb(Y)$ if the blank is at y-coordinate Y, and $adj(U,V)$ if U and V are adjacent x or y coordinates. The same goal now looks like

$$\{xloc(a,1), yloc(a,1), \ldots, \boxed{xlocb(2), ylocb(3)}\},$$

where x and y coordinates are numbered 1 through 3.

Two operators suffice for this representation:

$xmove(T,X,X')$ — move T from column X to X'

pre: $xloc(T,X), yloc(T,Y),$

$\boxed{xlocb(X'), ylocb(Y)}, adj(X,X')$

add: $xloc(T,X'), \boxed{xlocb(X)}$

del: $xloc(T,X), \boxed{xlocb(X')}$

$ymove(T,Y,Y')$ — move T from row Y to Y'

pre: $xloc(T,X), yloc(T,Y),$

$\boxed{xlocb(X), ylocb(Y')}, adj(Y,Y')$

add: $yloc(T,Y'), \boxed{ylocb(Y)}$

del: $yloc(T,Y), \boxed{ylocb(Y')}$

Other representations are also possible, for example, by treating the blank like a tile that happens to satisfy a predicate *isblank(T)*.

2.2 Abstraction

Transformation	English Paraphrase
drop_predicate(P)	drop all instances of predicate P from problem description
drop_goal(G)	drop subgoal G from goal
drop_precondition(P,O)	drop precondition P of op O
count(P)	replace predicate P by number of objects that satisfy it
compose_numbers(M,N)	replace two numbers by their sum
drop_number(N)	drop all instances of number N from problem description
number_to_parity(N)	replace number N with its parity
compose_parities(B,C)	replace two parities by their sum
drop_parity(B)	drop all instances of parity B from problem description

Table 1: Catalog of Abstracting Transformations

Given a particular problem class representation, ABSOLVER generates abstractions of it by applying transformations chosen from the catalog shown in Table 1. These transformations resemble generalisation rules like those in [Michalski, 1983], except that they apply to state-space problems rather than concept descriptions. For example, ABSOLVER can derive a heuristic that is functionally equivalent to Manhattan Distance by applying the *drop_predicate* transformation to the *blank* predicate in the single-operator representation, or to *xloeb* and *yloeb* in the Cartesian representation. In either case, the abstraction is formed by dropping the boxed predicates from the operators and goal. In the abstracted problem space, each tile can move to adjacent squares regardless of where the other tiles and blank are.

2.3 Optimization

Name	English Paraphrase
factor(P)	Factor problem P into independent subproblems
merge_operators(X,Y)	Merge two identical operators X and Y
collapse(Op)	Collapse non-branching search to closed form of cost

Table 2: Catalog of Optimizing Transformations

An abstracted problem can be sped up by one or more of the optimizing transformations listed in Table 2—provided it satisfies their preconditions. Unfortunately, most abstractions do not; Section 3.3 discusses the difficult problem of finding abstractions that can be optimized.

In our example, we can use the *factor* transforms tion to split the abstracted problem into 8 independent subproblems, one for each tile. Next, *factor* simplifies each subproblem by restricting the set of operators to those that are relevant to achieving the subgoal. For example, the first subproblem has the goal at(a, 1) and the restricted operator *move(a,* 5, *S').* For an n by n version of the Eight Puzzle, this step reduces the abstracted problem from a search space containing $(n^2 — l)^n$ states (n^2 possible locations for each tile) to (n^2 — 1) independently solvable problems of size n^2, which can be solved in total time $0(n^A)$.

If we use the Cartesian representation, we can obtain further speedup by applying *drop .precondition* to *xloc* in the *yrnove* operator and *yloc* in the *xmove* operator, and factoring into separate subproblems for the *x* and *y* dimensions. The resulting problem has $2n^2 — 2$ independent subproblems, one for each tile and each dimension. For example, the first subproblem has the goal *xloc(a,* 1) and the restricted operator *xmove(a, X, X').* Each subproblem has a search space of size n, corresponding to the number of columns (or rows). Thus the total time to evaluate Manhattan Distance is reduced to $0(n^3)$. In comparison, the standard closed-form formula for Manhattan Distance takes time $O(n^2)$, the number of tiles.

2.3.1 Testing Factorability

The *factor* transformation partitions the goal into mutually independent sets of subgoals, and identifies the operators relevant to each set. The independence property ensures that the sum of the optimal solution costs for achieving each set is optimal for achieving their union, and is therefore admissible.

Thus in order for *factor* to be practical, we need an efficient way to check that two sets of goals, g_1 and g_2, satisfy the following property:

$$(\forall s)\ dist(s, g_1 \cup g_2) \quad = \quad dist(s, g_1) + dist(s, g_2) \quad (1)$$

Here $dist(s, g)$ denotes the length of an optimal path from state s to goal g.

ABSOLVER computes a sufficient condition for (1) by overestimating the interactions among operators that might affect the various goals. The idea is that two goals are independent if no operator affects them both. To formalise this sufficient condition, we introduce the following relations:

Canhelp(g_1, g_2) iff some operator that adds g_1 also adds a precondition of some operator that adds (a conjunct of) g_2.

*Canhelp** is defined as the transitive closure of the Canhelp relation.

Affects(op, g) iff op adds or deletes a predicate p such that *Canhelp**(p, g).

Indep(g_1, g_2) iff $(\neg\exists op) Affects(op, g_1) \wedge Affects(op, g_2)$.

Indep(g_1, g_2) is the desired sufficient condition for (1).

To prove *Indep*(g_1, g_2), ABSOLVER computes the transitive closure of *Canhelp*. Without function symbols or negation, this closure is finite, but can be expensive to compute exactly. Therefore, ABSOLVER uses a form of symbolic evaluation that over-estimates the *Canhelp** relation. To form the set of goals that can help achieve a predicate p, ABSOLVER backchains through every operator that can add p, inserting its preconditions in the set. For example, when ABSOLVER backchains from the goal $at(a, 1)$ through the abstracted *move* operator, which adds $at(T, S')$, it binds T to a and forms the set $\{at(a, ANY), adj(ANY, ANY)\}$. This set is closed under the backchaining operation. A similar technique is used to identify the operators relevant to achieving a given set of goals. For the goals here, the only relevant operator is $move(a, S, S')$.

2.4 Using Heuristics

Since the cost-effectiveness of heuristics derived by ABSOLVER is generally difficult to predict, they must be evaluated empirically. ABSOLVER's performance element inputs an initial state, a problem to solve, and optionally a heuristic. The problem is represented as a list of one or more subproblems, each consisting of a subgoal and a set of operators with which to achieve it. The *IDA** search algorithm is used to solve each subgoal, guided by the heuristic (or breadth-first if none is given).

The heuristic is represented as an abstract problem. It is evaluated by recursively invoking the same search procedure on the abstract problem and returning the length of the optimal solution. Thus the computation of the heuristic can itself be guided by a hierarchy of successively more abstract problems.

To illustrate, consider the "X-Y Heuristic," which is derived by factoring an abstracted Cartesian representation of the Eight Puzzle into two subproblems, one for each dimension. However, instead of dropping the blank predicate to achieve factorability, the *drop-precondition* transformation is used to drop information about the X dimension from the subproblem for the Y dimension, and vice versa. In effect, each subproblem projects the puzzle onto one dimension. Thus a horizontal move is allowed only into the column containing the blank, and a vertical move is allowed only into the row containing the blank. X-Y is therefore more accurate than Manhattan Distance, which ignores the blank completely.

Although X-Y yields a base-level branching factor of only 1.19 for the Eight Puzzle (on a set of 18 random instances), it requires a considerable amount of search to compute. This search can be guided by Manhattan Distance. Unfortunately, even with such guidance the overall search time turns out to be about six times slower than using Manhattan Distance alone. It remains to be seen whether additional optimizations can make X-Y better than Manhattan Distance, which is the best known heuristic for Eight Puzzle.

3 Evaluating ABSOLVER

We now evaluate ABSOLVER from three perspectives: first, as an *explanatory model* that can rationally reconstruct existing heuristics, thereby verifying admissibility by construction; next, as a *generative model,* helpful for suggesting new heuristics; and finally, as an *automatic discovery engine.*

3.1 ABSOLVER as an Explanatory Model

Domain	Heuristic	Derivation
Eight Pussle	Manhattan Distance [Pearl, 1984]	drop *xlocb*, *ylocb*; *xloc* of *ymove*; *yloc* of *xmove*; factor tiles, X & Y
	n-MaxSwap [Gaschnig, 1979]	drop *adj*
	n-Swap [Gaschnig, 1979]	repr. blank as tile; drop *blank*, *adj*
	# Misplaced Tiles [Pearl, 1984]	drop *blank*, *adj*; factor tiles
	Euclidean Distance [Pearl, 1984]	based on geometry knowledge
	Sequence Score [Nilsson, 1980]	not derivable: not admissible
Towers of Hanoi	# Misplaced Disks [Amarel, 1983]	drop *clear*, *smaller*
	Alternating Disks [Berlekamp *et al.*, 1982]	not derivable: predicate on moves, not states
	Blocking Disks [Amarel, 1983]	not derived: based on algorithm
Mutilated Checker-Board	Colored Squares [Kibler, 1985] [Korf, 1980]	repr. square colors; count red & black; collapse to closed form

Table 3: Derivations of Known Heuristics

An explanatory model should be evaluated by its *generality* and *coverage.* A general model applies to a wide class of domains. Within that class, a good model covers a large proportion of the phenomena to be explained. We tested ABSOLVER's generality by applying it to several puzzle domains. We tested its coverage by trying to rederive all published heuristics for three of them: the Eight Pussle, Towers of Hanoi, and Mutilated Checkerboard. The results are summarized in Table 3.

This evaluation simultaneously tested the model at three different levels of specificity. First, it tested the generality and coverage of the particular catalog of transformations across several domains. Since we knew our initial catalog was incomplete, we were also interested in identifying useful new transformations. In fact, these problems served as our "training data" for developing the catalog in Table 1, so they should not be taken as a test of its coverage, though they do demonstrate the generality of the transformations used in multiple domains. Second, it tested the problem representation language, since some heuristics might not be derivable in that representation. Finally, it tested the general model of abstraction plus optimization, which might fail to explain some heuristics.

What does it mean to rederive a heuristic? Clearly, it is not necessary to rederive specific code. On the other hand, functional equivalence alone is insufficient, since an abstraction-based heuristic computed using search may be much less efficient than the original version. We therefore compared their computational complexity.

The results can be summarized as follows. We were able to rederive functional equivalents with ABSOLVER for 6 of the 10 published heuristics, though with varying efficiency relative to the original versions.

The derived versions of # of Misplaced Tiles, # of Misplaced Disks, and Colored Squares were all computationally equivalent to the originals.

Three of the derived heuristics were less efficient than the original versions, owing to limitations in the chosen problem representations and deficiencies in the catalog of optimizing transformations. The derived Manhattan Distance was slower by *0(n)* (for *n* by n puzzles) because it uses search to compute the number of moves needed to get from row % to row i''. We have derived the closed form expression *i* — i'| on paper, but only by exploiting the numerical relationships implicit in the *adj* relation to induce a recurrence relation. Implementing this derivation in ABSOLVER would require a more sophisticated representation and an optimizing transformation capable of inducing the recurrence relation. Similarly, the derived versions of n-MaxSwap and n-Swap are slower by $0(n^{2!})$ because they search for an ordered permutation instead of counting the number of swaps performed by an efficient sorting algorithm. A challenging direction for future work is automatically recognizing when an abstracted problem can be solved by adapting a known algorithm.

Two of the heuristics have the wrong form: Sequence Score is non-admissible, and Alternating Disks is a heuristic on moves, not states.

Euclidean Distance breaks our model in an interesting way, because it comes from adding knowledge about geometry. Our current model only generates abstractions by dropping information from the problem definition.

The Blocking Disks heuristic is defined only for states in which two of the pegs are empty. It was originally derived by analyzing a recursive algorithm for Towers of Hanoi to compute the exact number of moves required.

We see no way to derive this formula by abstracting the problem representation. However, extending ABSOLVER to find special-case heuristics might be worthwhile.

3.2 ABSOLVER as a Generative Model

Domain	Heuristic	Derivation
3x3x3 Rubik's Cube	Center-Corner	drop edges; factor into center & corner
Eight Puzzle	X-Y	drop *xloc*, *xlocb* of *ymove*; *yloc*, *ylocb* of *xmove*; factor into X & Y
	Distance of Blank	drop *xloc*, *yloc*; *xlocb* of *ymove*; *ylocb* of *xmove*; factor into X & Y
	# Out of Col. + # Out of Row	drop *adj*, *xlocb*, *ylocb*; *xloc* of *ymove*; *yloc* of *xmove*; factor into X & Y
Jagged Square	Border-Interior	count in partitions; merge operators
Fool's Disk	Diameter Sum	compose opposite radii sums
	Combined Diameters	compose diameter sums
Instant Insanity	Opposite Sides	repr. colors as #'s; compose opposite sides
Think-A-Dot	Dropped Gates	drop gate parities

Table 4: Novel Heuristics Derived

How good is our model at suggesting new heuristics? Table 4 attempts to answer this question for several domains. For each of these domains only "good" heuristics are listed; several other heuristics were derived but appeared worse in terms of overall search time.

We were able to discover the first known (non-trivial) admissible heuristic for the *3x3x3* Rubik's Cube. For this problem, we started with a represention that partitions the Cubies into center, edge, and corner ones. Dropping the edge predicate allows the operators to be factored into those that affect corner Cubies and those that affect center Cubies.

How good is this Center-Corner heuristic? For 13 problems randomly scrambled to depth 6 or less, it reduces the branching factor from 9 (for breadth-first search) to 5.9. The heuristic is computed by solving the two subproblems. The subproblem for the six center Cubies has three operators and is cheap to solve. The subproblem for the eight corner Cubies is equivalent to a 2x2x2 version of Rubik's Cube. Since our Prolog implementation of *IDA** only expands about 10 states per second, we have not yet evaluated the Center-Corner heuristic on deep random solvable instances of the Cube.

We were also able to derive some new admissible heuristics for the Eight Puzzle. We have already described the X-Y heuristic, which is more accurate than Manhattan Distance. Similarly, # Out of Column + # Out of Row is more accurate than # Misplaced Tiles, and takes only twice as long to compute.

We also used the techniques reported in this paper to help demonstrate the unsolvability of a once-open layout problem known as the Jagged Square Puzzle, shown in Figure 2. The task is to tile the 60-square figure using each pentomino once—or to prove that it cannot be done. When we attacked this problem, we believed it was still open. After proving it impossible, we found a published proof that relies on remarkably similar techniques [Golomb, 1965].

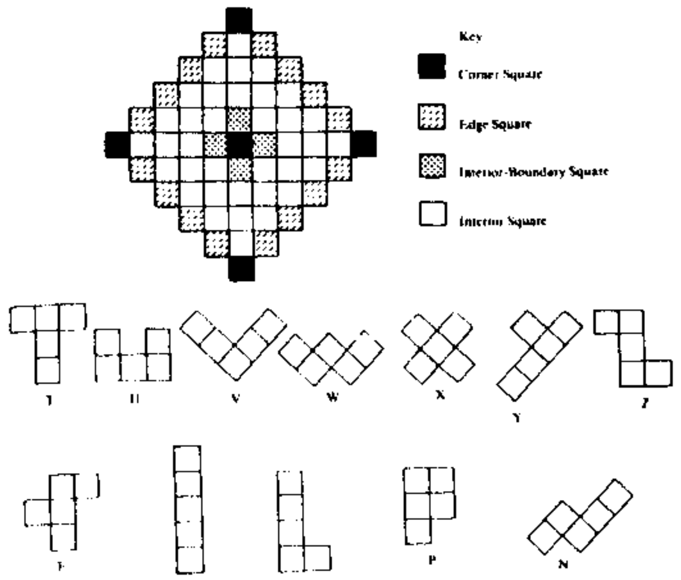

Figure 2: The Jagged Square Puzzle

We started with a problem representation that partitions the squares into those in the interior of the Jagged Square, those bordering the outer edges, and those bordering the inner "hole." (We later added "corners"). Next, we applied the *count* transformation to each partition. Unfortunately, it turned out that the resulting admissible heuristic was not strong enough to prove the problem impossible, i.e., the abstracted initial state appeared solvable. We then hand-generated every possible placement of the four most constraining pentominoes; taking symmetry into account, there were only a dozen or so. Finally, we executed exhaustive search in the abstract space on each of these partial layouts to show the impossibility of completing any of them.

Table 4 also lists novel heuristics derived for some GPS domains [Ernst and Goldstein, 1982]: Fool's Disk, Instant Insanity, and Think-A-Dot.

One lesson of these examples is the frequent importance of clever partitioning in the initial problem representation: the border and interior squares in Jagged Square; the corner, center, and edge Cubies in Rubik's Cube; and the red and black squares in Mutilated Checkerboard. At present such partitioning is supplied as part of the initial representation.

3.3 ABSOLVER as an Automatic Discovery Engine

As an automatic discovery engine, ABSOLVER must be evaluated by the *tractability* of finding good heuristics in the space defined by the catalog of transformations. There is a tradeoff between tractability and coverage, since enlarging the catalog expands the space of derivable heuristics but makes it costlier to explore.

The full space defined by ABSOLVER's current catalog of transformations is far too huge to explore exhaustively. For example, the *drop* transformations alone can be applied to any combination of subgoals and operator preconditions. For our nine-operator representation of the 3x3x3 Rubik's Cube, there are 126 subgoals and 312 preconditions: the number of combinations is astronomical.

As a compromise between coverage and tractability, we implemented an exhaustive generator using the coarser-grained *drop-predicate* as the only abstracting transformation and *factor* as the only optimizing transformation. For the Eight Puzzle, it took this generator a few CPU seconds to test all combinations of dropped predicates. For the three-predicate, single-operator representation, the only combinations with more than one factor yielded Manhattan Distance and # Out of Place Tiles (subject to the efficiency limitations discussed in Section 3.1). The five-predicate, two-operator Cartesian representation yielded more heuristics: Horizontal Distance, Vertical Distance, Manhattan Distance (their sum), analogs of all three for Distance of Blank, # Out of Column + # Out of Row, and Blank Out of Column + Blank Out of Row. For our three-predicate, nine-operator representation of Rubik's Cube, the generator took over five hours, since our independence test for factorability currently takes time quadratic in the size of the largest transitive closure formed by the symbolic backchaining step described in Section 2.3.1. Somewhat surprisingly, there is only one factorable combination—the Center-Corner heuristic.

To escape from the coverage-tractability tradeoff, we must use a better strategy than exhaustive generate-and-test to find efficiently computable abstractions. We arc investigating the use of means-ends analysis to identify which abstracting transformations will enable optimizations. In particular, if we can efficiently identify which applications of *dropsubgoal* and *drop_precondition* will make it possible to apply *factor,* we will be able to find the factorable abstractions without generating and testing all combinations of dropped goals and preconditions.

4 Relation to Previous Work

Figure 3 relates several previously reported properties that can hold between abstractions and heuristics [Nilsson, 1980, Pearl, 1984].

The relation of certain abstractions to state-space search heuristics was first suggested by Guida and Somalvico [Guida and Somalvico, 1979] and Gaschnig [Gaschnig, 1979], who described how such heuristics might arise by using the depth of solutions in edge supergraphs of the original state-space search graph as lower-bounding heuristics. Such edge supergraphs naturally arise from dropping operator preconditions. Later, Valtorta [Valtorta, 1984], Pearl IPearl, 1984], and Kibler [Kibler, 1985] each proved that abstractions such as those resulting from dropped operator preconditions would guarantee monotone (and hence admissible) heuristics. Our abstracting transformations extend the edge supergraph model of abstraction to include node-merging.

UPWARD SOLUTION PROPERTY
Abstracting any solution path gives a solution path for the abstracted problem [Tenenberg, 1988].

$\Uparrow$

ABSTRACTION:
Abstracted state space adds edges to the base problem space [Gaschnig, 1979] or merges nodes.

$\Downarrow$

MONOTONICITY:
$dist(s', g') \leq dist(s, t) + dist(t', g')$
(s', g', t' are abstractions of s,g,t; t is a descendant of s)

$\Downarrow$

ADMISSIBILITY:
$dist(s', g') \leq dist(s, g)$

$\Downarrow$

SOUNDNESS:
$Solvable(s, g) \Rightarrow Solvable(s', g')$

Figure 3: Properties of Abstracted State Space Problems

Valtorta proved that using a dropped-precondition abstraction directly as a heuristic will *always* expand more total states in the two spaces than simply using breadth-first search in the base space [Valtorta, 1984]. Pearl later pointed out that this liability might be overcome by factoring the abstracted problem into independent or serializable subproblems, which might be possible even when the original problem is not factorable [Pearl, 1984]. Factoring reduces the total search complexity from the product of exponentials to their sum. Though elegant, these methods were not implemented: the abstracting and optimizing transformations were performed by hand.

Some work has been done on automatic generation of abstractions in planning [Sacerdoti, 1974, Knoblock, 1988, Unruh *et* a*l*., 1987], but not for the class of admissible heuristics addressed here. For example, ABSTRIPS used abstract solutions as skeletons for base solutions, which tends to reduce planning time but can produce sub-optimal plans. Furthermore, ABSTRIPS discards abstract solutions that cannot be refined into more concrete ones. In contrast, ABSOLVER uses abstractions solely to compute lower bounds and check solvability, thereby not discarding potentially valuable information from non-refinable abstract solutions. Other systems have been reported for serializing GPS subgoals [Ernst and Goldstein, 1982], but they do not guarantee the optimally of the solution paths.

In sum, while a few techniques have previously been reported for abstracting and simplifying state-space problems, ABSOLVER constitutes a novel attempt to *automate, integrate, extend,* and *evaluate* these techniques.

5 Conclusion

ABSOLVER appears to be the first mechanical generator of state-space heuristics guaranteed to find optimal solution paths. It achieves this admissibility property by decomposing the problem of discovering heuristics into generating abstractions and optimizing their evaluation.

ABSOLVER can be viewed at more than one level. First, its transformational model provides a unifying framework for characterizing and exploring a broad class of admissible heuristics and understanding when they are actually useful. Second, we have *grounded* the model by implementing a catalog of abstracting and optimizing transformations and using them to derive a number of heuristics. Third, we have demonstrated an automatic generator that uses two of these transformations to find efficiently computable heuristics.

As an explanatory model, ABSOLVER's coverage is encouraging in one sense but deficient in another. While its small catalog of abstracting transformations is adequate to derive functional equivalents for many of the published heuristics we looked at in several puzzle domains, its optimizing transformations are too weak to compute some of them as efficiently as the originals. We do not claim the catalog is complete, and in fact expect it to grow as we try to improve evaluation cost, derive more heuristics, and explore other domains. The few heuristics that did not match ABSOLVER's underlying model suggest interesting directions in which to extend it.

As a generative model, ABSOLVER has yielded some novel heuristics, notably the first non-trivial admissible heuristic for Rubik's Cube, and an interesting Eight Puzzle heuristic that is more accurate than the best known, though somewhat less cost-effective. Although all the transformations reported here are fully implemented, the techniques used in ABSOLVER are useful for generating heuristics even when applied by hand. In fact, that is how we actually discovered most of the novel heuristics, and how we proved the impossibility of a published layout problem we thought was still open. Subsequent implementation of the derivations served to verify their correctness and to expose the use of additional techniques. Thus the implementation is actually of secondary importance for discovering new heuristics, except to the extent that it makes the techniques easier to apply.

As an automatic discovery engine, ABSOLVER is limited by the intractability of exploring the space generated by its full catalog of transformations. Its exhaustive generator uses only one abstracting transformation and one optimizing transformation, thereby achieving tractability at the cost of coverage. Nonetheless, it finds interesting heuristics in more than one domain.

Of the many possible directions for extending ABSOLVER, two seem especially compelling. First, means-ends analysis may make it possible to explore a richer space of possible heuristics automatically. Second, ABSOLVER's sensitivity to the initial problem representation, and the importance of clever partitioning in discovering novel heuristics, suggest that a few transformations for representation-shifting might significantly enrich the space of discoverable heuristics.

References

[Amarel, 1983] S. Amarel. Representations in problem-solving. In *Methods of Heuristics.* Lawrence Erlbaum and Associates, Palo Alto, CA, 1983.

[Berlekamp *et al,* 1982] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways for your Mathematical Plays: Volume II.* Academic Press, London, 1982.

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

[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.

[Golomb, 1965] S. Golomb. *Polyominoes.* Charles Scribners and Sons, New York, 1965.

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

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

[Knoblock, 1988] Craig A. Knoblock. Automatically generating abstractions for planning. In *Proceedings of the First International Workshop in Change of Representation and Inductive Bias,* Briarcliff, NY, 1988. Philips Laboratories.

[Korf, 1980] R. Korf. Towards a model of representation changes. *Artificial Intelligence,* 14(l):41-78, 1980.

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

[Korf, 1985b] R. Korf. *Learning to Solve Problems by Searching for Macro-Operators.* Pitman, Marshfield, MA, 1985.

[Michalski, 1983] R.S. Michalski. A theory and methodology of inductive learning. In *Machine Learning,* pages 83-134. Palo Alto, CA: Tioga Publishing Company, 1983.

[Nilsson, 1980] N. J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufman n, Palo Alto, CA, 1980.

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

[Sacerdoti, 1974] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence,* 5:115-135, 1974.

[Tenenberg, 1988] J. Tenenberg. *Abstraction in Planning.* PhD thesis, University of Rochester, 1988.

[Unruh *et ai,* 1987] A. Unruh, P. Rosenbloom, and J. Laird. Dynamic abstraction problem-solving in Soar. In *Proceedings of the Third Annual Conference on Aerospace Applications of Artificial Intelligence,* Dayton, OH, October 1987.

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