Learning Abstraction Hierarchies for Problem Solving

Craig A. Knoblock*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
cak@cs.cmu.edu

Abstract

The use of abstraction in problem solving is an effective approach to reducing search, but finding good abstractions is a difficult problem, even for people. This paper identifies a criterion for selecting useful abstractions, describes a tractable algorithm for generating them, and empirically demonstrates that the abstractions reduce search. The abstraction learner, called ALPINE, is integrated with the PRODIGY problem solver [Minton *et al.*, 1989b, Carbonell *et al.*, 1990] and has been tested on large problem sets in multiple domains.

Introduction

Hierarchical problem solving uses abstraction to reduce the complexity of search by dividing up a problem into smaller subproblems [Korf, 1987, Knoblock, 1989b]. Given a problem space and a hierarchy of abstractions, called abstraction spaces, a hierarchical problem solver first solves a problem in an abstract space, and then uses the abstract solution to guide the search for a solution in successively more detailed spaces. The technique was first used in GPS [Newell and Simon, 1972] and has since been used in a number of problem solvers. ABSTRIPS [Sacerdoti, 1974] was the first system that attempted to automate the formation of abstraction spaces, but only partially automated the process. Most hierarchical problem solvers are simply provided with abstractions that are hand-tailored to a specific domain [Sacerdoti, 1977, Tate, 1977, Wilkins, 1984].

This paper describes an abstraction learner, called ALPINE, that completely automates the formation of abstraction hierarchies. Given a problem space, which consists of a set of operators with preconditions and effects, and a problem to be solved, ALPINE reformulates this space into successively more abstract ones. Each abstraction space is an approximation of the original problem space (base space), formed by dropping literals in the domain. The system determines what to abstract based on the *ordered monotonicity property*. This property separates out those features of the problem that can be solved and then held invariant while the remaining parts of the problem are solved. Since this property depends on the problem to be solved, ALPINE produces abstraction hierarchies that are tailored to the individual problems.

The paper is organized as follows. The next section defines the ordered monotonicity property. The third section describes the algorithm for generating abstraction hierarchies in ALPINE. The fourth section describes the use of the abstractions for hierarchical problem solving, which is implemented in PRODIGY. The fifth section presents extensive empirical results that demonstrate the effectiveness of the abstractions in reducing search. The sixth section outlines the related work on automatically generating abstractions. The last section summarizes the contributions and sketches some directions for future work.

Ordered Abstraction Hierarchies

This section defines the *ordered monotonicity* property and describes how this property can be used as the basis for generating useful abstraction hierarchies. The ordered monotonicity property captures the idea that as an abstract solution is refined, the structure of the abstract solution should be maintained. The process of refining an abstract plan requires inserting additional steps to achieve the literals (possibly negated atomic formula) ignored at the abstract level. The property constrains this refinement process.

The ordered monotonicity property of an abstraction hierarchy is defined as follows. (A more formal definition of this property can be found in [Knoblock, 1990].)

Ordered Monotonic Refinement: A refinement of an abstract plan that leaves the truth value of every

*The author is supported by an Air Force Laboratory Graduate Fellowship through the Human Resources Laboratory at Brooks Air Force Base. This research was sponsored in part by the Office of Naval Research under contract number N00014-84-K-0415, and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499.

literal in an abstract space unchanged.[1]

Ordered Monotonic Hierarchy An abstraction hierarchy with the property that for every solvable problem there exists an abstract solution that has a sequence of ordered monotonic refinements into the base space.

(Throughout the remainder of the paper *ordered* is used to mean *ordered monotonic*.) An ordered hierarchy can be constructed by dividing the literals that comprise a domain into levels (abstraction spaces) such that the literals in a given level do not interact with any literals in a more abstract level. If a solution exists it will be found by searching for abstract solutions and ordered refinements of those solutions. If there is no abstract solution, the problem is unsolvable. Unfortunately, there is no guarantee that an ordered abstraction hierarchy will reduce the overall search, since it may be necessary to backtrack to find alternative abstract solutions. In practice, this property provides a surprisingly good criterion for producing useful abstraction hierarchies.

Consider an example that distinguishes an ordered abstraction hierarchy from an unconstrained one. A simple machine-shop scheduling domain consists of operations for changing the shape, making holes, and painting parts. The operations interact in various ways. For example, changing the shape of a part also removes any holes or paint. A problem solver is given the problem of producing a black, cylindrical part with a hole drilled through it. Imagine that the problem solver uses an abstraction that ignores shape and produces an abstract plan that drills the hole and paints the part black. The plan is then refined in the next abstraction level to make the part cylindrical, but when this step is inserted in the abstract plan it changes the truth values of the literals involving both holes and paint that were achieved in the abstract space. The resulting refined plan might be: drill hole, paint black, make cylindrical, drill hole, paint black. This is a valid refinement in the sense that it achieves the goal, but not an ordered refinement because it altered the paint and hole properties, which had already been achieved in a more abstract space. The ordered monotonicity property requires that the operation of making the shape of a part cylindrical is placed at the same or more abstract level as the other two operators because changing the shape undoes the effects of the other operators. If shape were dealt with first, the problem solver would produce an abstract plan that made the part cylindrical and then insert the steps to make the hole and paint the object.

[1] An ordered monotonic refinement is more restrictive than a monotonic refinement [Knoblock, 1989b] because it requires that every literal in the abstract space is left unchanged instead of just the specific literals that comprise the abstract plan. The distinction between an ordered monotonic hierarchy and a monotonic one is analogous.

Using the ordered monotonicity property, ALPINE produces problem-specific abstraction hierarchies. The standard approach to using abstraction spaces is to provide a system with a single, fixed abstraction hierarchy, which is then used for all problem solving in a given domain. The disadvantage of this approach is that it limits the possible abstractions in a domain since the hard parts of one problem may be details in another problem and vice versa. Instead of attempting to find a fixed abstraction hierarchy to use for all problems, ALPINE dynamically generates an abstraction hierarchy that is tailored to the particular problem to be solved. The next section describes how ALPINE generates these abstraction hierarchies.

Learning Abstraction Hierarchies

ALPINE is a fully implemented abstraction learner, which produces abstraction hierarchies for use in the PRODIGY problem solver. The system is given a domain specification, which consists of the operators and axioms that define a problem space. For each problem to be solved, ALPINE selects an abstraction hierarchy and uses that hierarchy for problem solving.

ALPINE forms abstraction hierarchies by grouping the literals in a domain into classes and ordering the classes. Initially, literals of the same type are placed in the same class (e.g., `(SHAPE A CYLINDRICAL)` and `(SHAPE B RECTANGULAR)` are both instances of the literal type `(SHAPE obj shape)`). The initial classes are further combined and then ordered based on an analysis of the domain. The abstraction spaces are then formed by successively removing the literal classes from the original problem space. An abstract space does not simply involve dropping preconditions or goals; instead an abstract space is an abstract model of the original problem space, where both the operators and states are simplified.

Table 1 defines the algorithm for creating a problem-specific abstraction hierarchy. The first step in the algorithm produces a set of constraints on the order of the literals in an abstraction hierarchy. The constraints are placed in a directed graph, where the literals form the nodes and the constraints form the edges. A constraint in the graph indicates that some literal must be higher (in a more abstract space) or at the same level in the abstraction hierarchy as some other literal. The constraints are sufficient to guarantee that a hierarchy built from these constraints will have the ordered monotonicity property. The algorithm for determining these constraints is described below. The second step in the procedure finds the strongly connected components and combines the literal classes within each connected component. Each strongly connected component contains the literals that comprise an abstraction space, and the constraints between components determines a partial order of abstraction spaces. The third step in the procedure performs a topological sort of the strongly connected components to find a total ordering

of the literal classes, which forms an abstraction hierarchy. Efficient algorithms for determining the strongly connected components and performing a topological sort can be found in [Aho *et al.*, 1974].

Input: Domain operators and a problem to be solved.
Output: An ordered hierarchy for the given problem.

```
procedure Create_Hierarchy(goal,operators):
1.  Find_Constraints(goal,operators);
2.  Combine_Strongly_Connected_Components(GRAPH);
3.  Topological_Sort(GRAPH)
```

Table 1: Creating an Abstraction Hierarchy

Table 2 defines a recursive procedure for finding a sufficient set of constraints to guarantee the ordered monotonicity property. Lines 1-4 loop through each of the literals in the goal and find the operators that can be used to achieve these literals. An operator can only be used to achieve a literal if the literal is in the primary effects of the operator, which is part of the operator definition. Lines 5-6 add constraints that force the other effects of those operators to be at the same or lower level in the abstraction hierarchy as the goal literals. Lines 7-10 determine the preconditions of the operator that could be subgoaled on and add constraints that force these potential subgoals to be at the same or lower level in the hierarchy as the goal literals. Lastly, line 11 calls the procedure recursively on the potential subgoals.

Input: Domain operators and a problem to be solved.
Output: A sufficient set of constraints to guarantee an ordered abstraction hierarchy for the given problem.

```
procedure Find_Constraints(goal_lits,operators):
1.for each lit in goal_lits do
2.   if Constraints_Not_Determined(lit,GRAPH) then
3.     for each op in operators do
4.       if lit in Primary_Effects(op) do {
5.         for each other_lit in Effects(op) do
6.           Add_Directed_Edge(lit,other_lit,GRAPH);
7.         preconds ← Preconditions(op);
8.         subgoals ← Can_Be_Sugoaled_On(preconds);
9.         for each prec_lit in subgoals do
10.          Add_Directed_Edge(lit,prec_lit,GRAPH);
11.        Find_Constraints(subgoals,operators) }
```

Table 2: Finding the Constraints on a Hierarchy

The procedure for determining the potential subgoals, `Can_Be_Subgoaled_On`, returns only those preconditions that could be subgoaled on while solving the given problem. The naive approach is simply to mark every precondition that can be achieved by some operator as a potential subgoal. However, this would overconstrain the final abstraction hierarchy since there are preconditions of an operator will never be subgoaled on and there are other preconditions that would not be subgoaled on in the context of achieving particular goals. Instead, ALPINE determines the subgoals based on the goal context and some simple domain axioms. This analysis is performed in a preprocessing step that only needs to be done once for a domain. When an hierarchy is created the algorithm simply looks up the potential subgoals in a table. This step is completely described in [Knoblock, 1990].

When there are no abstractions for a problem the directed graph will collapse into a single node. An important advantage of the problem-specific abstractions is that the algorithm will produce fewer constraints, which reduces the likelihood that the hierarchy will collapse.

Consider a problem in the scheduling domain that requires making a part cylindrical, painting it black, and joining it to another part. Each of these top-level goals would generate constraints on the final abstraction hierarchy. For example, a part can be made cylindrical using the lathe operator, which has the side effect of removing any paint. Thus a constraint would be generated that forces **shape** to be higher or at the same level in the hierarchy as **painted**. A part can be joined to another part by bolting the two parts together, and the bolt operator has a precondition that the two parts have holes. Thus another constraint is generated that forces **joined** to be higher or at the same level as **has-hole**. This process continues until all the required constraints have been generated. The final directed graph for this problem is shown in Figure 1. The dotted boxes indicate the connected components and the arrows indicate the constraints.

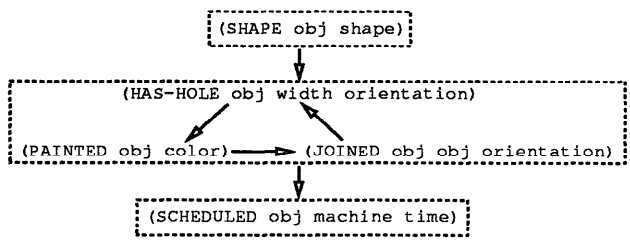

Figure 1: Directed Graph for a Scheduling Problem

Each abstraction hierarchy generated by the algorithm above has the ordered monotonicity property. The constraints guarantee that once the literals in a given space are achieved they cannot be undone in the process of refining the plan. This holds because the constraints force any literal that could be changed, either directly or indirectly in the process of achieving some goal, to be placed at the same or lower level in the abstraction hierarchy as the goal literal. The proof

that the basic algorithm produces ordered abstraction hierarchies is given in [Knoblock, 1989b].

The complexity of determining the constraints, and thus the complexity of creating the problem-specific abstraction hierarchies, is $O(o * n^2)$, where o is the maximum number of operators relevant to achieving any given literal and n is the number of different types of literal in the domain. The complexity of the other steps in creating the abstraction hierarchies, combining the strongly connected components and performing the topological sort, are quadratic in the number of literal types. Thus, for reasonable values of n, it is tractable to produce problem-specific abstraction hierarchies.

Hierarchical Problem Solving

To plan hierarchically, a problem is first mapped into the highest level of the abstraction hierarchy. This is done by removing details from the initial and goal states to form the corresponding abstract states and removing details from the preconditions and effects of the operators to form a set of abstract operators. Next, the problem is solved in this abstract problem space. The resulting abstract plan is then mapped into successively more detailed levels by forming subproblems where each intermediate state in an abstract plan forms an intermediate goal at the next level of detail. When the problem has been refined into the base space (the original problem space), the problem is solved. If an abstract plan cannot be refined (e.g., conditions introduced at the current abstraction level cannot be satisfied), then the problem solver backtracks to reformulate the plan at a higher level of abstraction.

The hierarchical problem solving is implemented in the PRODIGY architecture [Carbonell *et al.*, 1990]. PRODIGY is a means-ends analysis problem solver, which has been extended to perform the necessary hierarchical control and bookkeeping. The system can backtrack efficiently both across abstractions levels and within an abstraction level by maintaining the relevant problem-solving traces. While PRODIGY makes effective use of alpine's abstraction spaces, they are sufficiently general to be used by any hierarchical problem solver that employs a state-space representation.

Results

ALPINE produces useful abstraction hierarchies in a number of problem domains. This section demonstrates the effectiveness of ALPINE's abstractions in a machine-shop scheduling domain and a robot planning domain [Minton, 1988, Minton *et al.*, 1989a]. These domains were originally used to evaluate explanation-based learning (EBL) in PRODIGY. A problem in the machine-shop scheduling domain involves finding a valid sequence of machining operations and scheduling the operations to produce various parts. The robot planning domain is an extended version of the STRIPS domain [Fikes and Nilsson, 1971], which includes locks, keys, and a robot that can both push and carry objects.

ALPINE produces useful abstraction hierarchies in both problem domains. In the scheduling domain, the abstraction hierarchies provide an order on the operations that need to be performed and separate the planning of the operations from the scheduling. In the robot planning domain the movement of boxes, keys, and the robot are separated from the details.

To evaluate the abstraction hierarchies produced by ALPINE, this section compares problem solving with ALPINE's abstractions to problem solving in PRODIGY. In the scheduling domain, the use of ALPINE's abstractions is also compared to the use of search control rules generated using explanation-based learning [Minton, 1988]. The systems were tested on 100-200 randomly generated problems in each domain and were allowed to run on each problem until it was solved or the 300 second CPU time limit was exceeded. Since ALPINE uses problem-specific abstraction hierarchies, the time spent creating the abstraction hierarchy for each problem (between 0.5 and 3.5 seconds) was included in the cost of solving a problem.

Figure 2 compares ALPINE, PRODIGY, and EBL in the scheduling domain. The graph shows the cumulative CPU time (in seconds) on all solved problems up to the given problem number. The systems were run on the 100 problems that were originally used for testing EBL, but the graph only includes the problems that could be solved by all three systems within the time limit. The number of problems that could not be solved by each system is shown in the key. Since the problems are ordered by increasing size, the cumulative time curves upward. The graph shows that ALPINE and EBL produce comparable speedups, and both systems produce significant speedups over the basic problem solver. In addition, ALPINE solved 98% of the problems within the time limit, while EBL solved 94%, and PRODIGY solved only 76%. As discussed later, future work will explore the integration of abstraction and EBL.

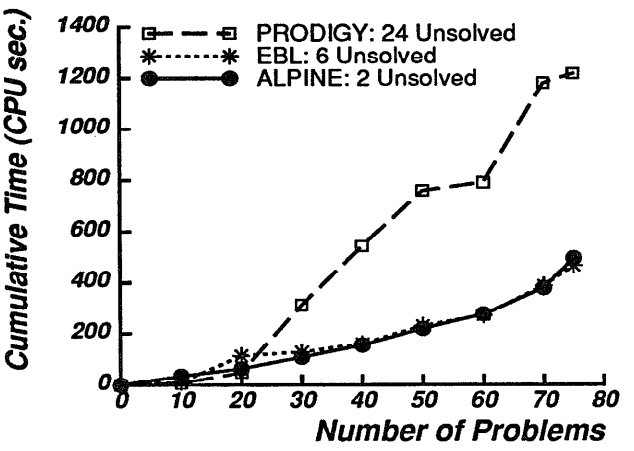

Figure 2: Comparison in the Scheduling Domain

Figure 3 shows an analogous graph that compares ALPINE and PRODIGY in the robot planning domain. EBL was not included in this comparison because ALPINE uses knowledge about the primary effects of the operators to produce the abstractions in this domain, and EBL was not originally provided with this knowledge. To avoid an unfair bias in the favor of ALPINE, both ALPINE and PRODIGY are given control knowledge to exploit the information about primary effects. The systems were run on 200 problems, including the 100 problems used for testing EBL. As the graph shows, ALPINE performed significantly better than PRODIGY on the solvable problems, and ALPINE was able solve 99% of the problems within the time limit, while PRODIGY only solved 90%.

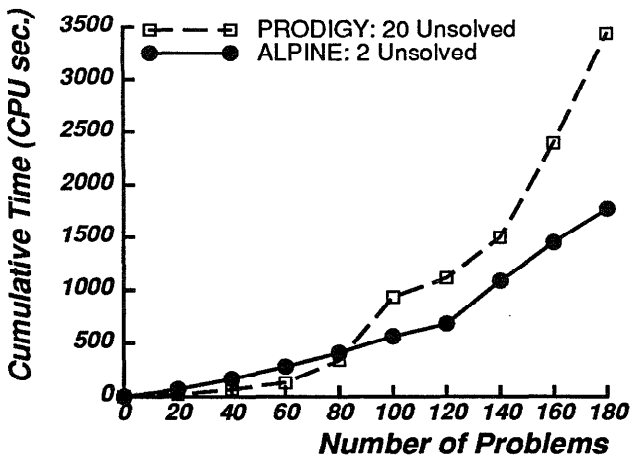

Figure 3: Comparison in the Robot Planning Domain

Related Work

ABSTRIPS [Sacerdoti, 1974] was one of the earliest attempts at automating the formation of abstraction hierarchies. The system was provided with an initial abstraction hierarchy, which was used to automatically assign criticalities to the preconditions of the operators. ABSTRIPS formed the abstraction levels by placing those preconditions which could not be achieved by a "short plan" in a separate level. Any further refinement of the levels came from the user-defined abstraction hierarchy. In the ABSTRIPS's domain, ALPINE completely automates the formation of the abstraction hierarchies and produces abstractions that are considerably more effective at reducing search [Knoblock, 1989a].

Unruh and Rosenbloom [1989] describe a weak method implemented in SOAR that dynamically forms abstractions for look-ahead search by ignoring unmatched preconditions. The choices made in the look-ahead search are stored by SOAR's chunking mechanism and the chunks are then used to guide the search in the original space. This approach forms abstractions based on which conditions hold during problem solving, while ALPINE forms abstractions based on the structure of the domain,

PABLO [Christensen, 1990] is a hierarchical planner that also forms its own abstraction hierarchies, but the system uses a completely different approach from the one described in this paper. The operators are partially evaluated before problem solving to determine the number of steps required to achieve any given goal. The system then solves a problem in successive abstraction levels by first working on the parts of the problem that require the greatest number of steps.

GPS did not create abstractions, but did automatically generate difference orderings [Eavarone, 1969, Ernst and Goldstein, 1982], which specify the order in which to work on the various goal conditions. The algorithm described in this paper is similar to the techniques for ordering differences, but the difference ordering algorithm only considers the interactions between effects of operators, while the algorithm described in this paper considers the interactions between both effects and preconditions of operators.

Mostow and Prieditis [1989] identify a set of transformations that can be used to form admissible heuristics. One of these is the *drop_predicate* transformation, which produces abstract problem spaces. Since the abstractions are formed using a brute-force generate and test procedure, the techniques described in this paper could be applied to their work.

Discussion

This paper described a general criterion for selecting abstraction hierarchies and presented a tractable algorithm for generating them. The approach has been applied to a variety of problem-solving domains and, as shown in this paper, produces abstractions that are effective at reducing search.

A current limitation of ALPINE is that the granularity of the abstractions is limited to the literal types in a domain. To address this problem the system has been extended to abstract specific instances of literals. Thus, instead of dropping all the literals of the same type, the system can drop instances of literals that can been shown not to interact with different instances of the same literal (e.g., `(SHAPE A CYLINDRICAL)` could be separated from `(SHAPE B RECTANGULAR)`). The difficulty is that the cost of forming these finer-grained abstraction hierarchies becomes prohibitive as the number of literal classes gets large. Future work will include developing techniques to efficiently generate these finer-grained hierarchies.

Another promising direction for future work is the integration of ALPINE with other types of learning, such as explanation-based learning and learning by analogy [Veloso and Carbonell, 1989]. Since ALPINE forms abstract models of the original problem space, these other types of learning can be applied within the abstract spaces. Thus, EBL could produce control knowledge

within an abstract space, and analogy could store and reuse abstract problem-solving episodes.

Acknowledgements

I am grateful to Jaime Carbonell, James Altucher, Oren Etzioni, Steve Minton, Josh Tenenberg, and Qiang Yang for their insights on the work presented in this paper. I would also like to thank Claire Bono, Dan Kahn, Alicia Perez, and Manuela Veloso for their comments on earlier drafts of this paper.

References

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[Carbonell *et al.*, 1990] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1990. Also Technical Report CMU-CS-89-189.

[Christensen, 1990] Jens Christensen. A hierarchical planner that creates its own hiearchies. In *Proceedings of Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Eavarone, 1969] Daniel S. Eavarone. A program that generates difference orderings for GPS. Technical Report SRC-69-6, Systems Research Center, Case Western Reserve University, Cleveland, OH, 1969.

[Ernst and Goldstein, 1982] George W. Ernst and Michael M. Goldstein. Mechanical discovery of classes of problem-solving strategies. *Journal of the Association for Computing Machinery*, 29:1–23, 1982.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Knoblock, 1989a] Craig A. Knoblock. Learning hierarchies of abstraction spaces. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 241–245, Los Altos, CA, 1989. Morgan Kaufmann.

[Knoblock, 1989b] Craig A. Knoblock. A theory of abstraction for hierarchical planning. In Paul Benjamin, editor, *Proceedings of the Workshop on Change of Representation and Inductive Bias.* Kluwer, Boston, MA, 1989.

[Knoblock, 1990] Craig A. Knoblock. *Learning and Using Abstractions for Hierarchical Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990. In preparation.

[Korf, 1987] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[Minton *et al.*, 1989a] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40(1-3):63–118, 1989.

[Minton *et al.*, 1989b] Steven Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1988.

[Mostow and Prieditis, 1989] Jack Mostow and Armand E. Prieditis. Discovering admissible heuristics by abstracting and optimizing: A transformational approach. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 701–707, San Mateo, CA, 1989. Morgan Kaufmann.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Sacerdoti, 1977] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, New York, 1977.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–900, 1977.

[Unruh and Rosenbloom, 1989] Amy Unruh and Paul S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 681–687, San Mateo, CA, 1989. Morgan Kaufmann.

[Veloso and Carbonell, 1989] M. M. Veloso and J. G. Carbonell. Learning analogies by analogy - the closed loop of memory organization and problem solving. In *Proceedings of the Second Workshop on Case-Based Reasoning*, pages 153–159, Los Altos, CA, May 1989. Morgan Kaufmann.

[Wilkins, 1984] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984.