Learning Subgoal Sequences for Planning

David Ruby*
Dennis Kibler
Information & Computer Science
University of California, Irvine
Irvine, CA 92717 U.S.A.
druby@ics.uci.edu

Abstract

A learning problem solver consists of three components: (1) a problem solver, (2) a memory of problem-solving knowledge, and (3) a learning component for deriving new problem-solving knowledge from experience. Previous learning problem solvers have acquired knowledge as macros, control knowledge, or cases. SteppingStone is a general learning problem solver that improves its performance by learning subgoal sequences. The underlying problem solver for SteppingStone is a combination of means-ends analysis and brute-force search. It depends primarily upon means-ends analysis and its problem-solving knowledge to solve the subgoals of the problem. Learning occurs only when this approach fails. Upon failure, search is applied and a new subgoal sequence is derived and added to its problem-solving knowledge. Before SteppingStone attempts to solve any of the problem subgoals, it first orders them with a domain independent heuristic which we call *openness.* Openness is used to order the subgoals to minimize interactions. SteppingStone's ability to improve its performance and scale to difficult problems is demonstrated with an implemented system. We show that a small memory of appropriate subgoals yields multiple orders of magnitude savings in problem-solving cost.

1 Introduction

Early planners were capable of solving only the simplest problems. In response, researchers sought ways of improving problem-solving performance by augmenting the planner with knowledge gleaned from experience. One approach acquires new operator sequences or macros [Fikes *et* a/., 1972, Korf, 1985b, Minton, 1985, Iba, 1985]. A second approach learns control knowledge, either represented as heuristics [Kibler, 1984, Pearl, 1984], or production rules [Mitchell *et* a/., 1983, Langley, 1985, Porter and Kibler, 1986, Minton, 1988].

*This work was partially supported by a grant from the Hughes Artificial Intelligence Center.

Both of these approaches are encompassed by a general model of human learning called chunking [Laird *et* a/., 1986]. A third approach to learning from experience, learning problem-solving cases, operates through storage and reuse of previous problem-solving sessions. These previous sessions are incorporated into a global knowledge structure along with any other knowledge about the world and planning, and are indexed by their goals solved, impasses encountered, failures encountered or similarity to the current problem [Hammond, 1986, Alterman, 1986, Carbonell, 1986, Kolodner, 1987, Bradtke and Lehnert, 1988, Ruby and Kibler, 1988].

We present a new representation for problem-solving knowledge, sequences of intermediate subgoals (stepping stones) between problem and goal states. Learning sequences of subgoals has the effect of decreasing the distance between problem states and goals without increasing the branching factor. Learning macros also decreases the distance between problem states and goals, but it increases the branching factor. Learning control knowledge decreases the branching factor of the problem, but leaves the distance between problem states and goals unchanged.

The value of subgoals has been analytically demonstrated by Korf [1988]. For example, if k appropriate intermediate subgoals are learned for a problem and a brute-force search technique is used, then the amount of search required to solve the problem decreases from b^d to $kb^d/^k$, where b is the branching factor of the domain and d is solution depth. In addition, if a problem can be broken into k independent subgoals, the branching factor b of the problem can be decreased by a factor of k. Clearly subgoals have immense value. The problem is how to learn thern.

SteppingStone is a general problem solver which learns to improve its performance by learning sequences of subgoals. It learns these new subgoal sequences when normal problem solving fails. Search is applied during these situations and the results are used to generate a new sequence of subgoals for memory. By restricting learning to those situations where normal problem solving fails, the utility of the learned knowledge is kept high.

2 Learning Subgoal Sequences

A learning problem solver consists of three basic components: (1) a problem solver, (2) a memory of problem-

solving knowledge used to assist the problem solver, and (3) a learning component that is able to compile experience into additional problem-solving knowledge. PRODIGY [Minton, 1988J and SOAR [Laird *et* a/., 1986] both use weak methods as their basic problem solvers. Prodigy learns control rules to assist its means-ends problem solver. SOAR learns production rules to record the solution to impasses. SteppingStone combines means-ends analysis and brute-force search for its problem solver. Its means-ends system does not allow undoing any previously solved subgoals unless the current subgoal explicitly directs it. Initially, SteppingStone depends upon its means-ends system and knowledge of subgoals sequences to solve the problem, but resorts to search when this fails. The results of the search are generalized, not into a macro or control rule, but into a sequence of subgoals that can be reused by the means-end system.

SteppingStone, like SOAR and Prodigy, uses a state-space representation of a problem, consisting of a set of operators, a start state, and a goal state. The goal state is represented as a conjunction of subgoals. The following pseudo-code outlines the SteppingStone approach:

```
ORDER problem subgoals
Loop through ordered problem subgoals
  Apply means-ends to current subgoal
  If means-ends fails but memory has solution Then
    Apply SteppingStone to subgoals from memory
  Elseif means-ends and memory fails Then
    Apply search to failure
    LEARN new subgoal sequence and store in memory
End loop
```

The following sections will further elaborate this description, with particular attention to the method for ordering the problem subgoals and learning new subgoal sequences.

2.1 Goal Ordering Based on Openness

SteppingStone begins by ordering the subgoals of the problem so few impasses are encountered. This ordering is found using a domain independent heuristic we call *openness.* Given a set of goals, some of which have already been solved, openness measures the likelihood of solving the remaining goals while preserving the solved goals.

More formally, openness is defined first for a single unsolved goal, and a set of solved goals. Let *G* denote a set of solved goals, G_2 a set of unsolved goals, and *g* a single goal in G_2 The openness of *g* with respect to G_1 is defined as:

Openness(Gi, *g)* = number of operators solving *g* whose preconditions are not contradicted by the assertion of goals G_1.

Intuitively, this measures the likelihood of moving from a state where the goals *G* are solved but *g* is not solved, to a state where both G_1 and *g* are solved. The openness of the set G_2 of unsolved goals with respect to the set *G* of solved goals is now defined as:

$$\text{Openness}(G_1, G_2) = \sum_{g \in G_2} \text{Openness}(G_1, g)$$

Given a goal set *G* consisting of *g*, g_2,..., g_n, the desired ordering maximizes the sum:

$$\sum_{i=1}^{n} \text{Openness}(g_1 \ldots g_i, g_{i+1} \ldots g_n).$$

Since there are n! factorial orderings of these goals, an exhaustive search is not usually possible. Instead, an ordering is built up iteratively by beginning with those goals *g £ G* that maximize Openness(g, *G — g).* Here G — *g* includes those goals in the set *G* except the goal *g.* This selection generates initial partial orderings consisting of an ordered set Gi, and unordered set G_2. These partial orderings are then iteratively expanded by adding goals *g* € G_2 that maximize Openness(G_1 + *g,G*$_2$ — *g)*> The set G_1 + *g* includes goal *g* along with all those goals in *G*. For each iteration *k,* only those partial orderings with maximum sums

$$\sum_{i=1}^{k} \text{Openness}(g_1 \ldots g_i, g_{i+1} \ldots g_n)$$

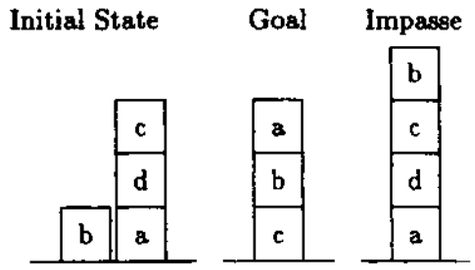
are further expanded. Although this search can be expensive, experimentation has proved it tractable (see Table 2). The final ordering of the goals minimizes the liklihood of generating an impasse during problem solving by maximizing the probability of solving the problem subgoals.

2.2 Memory of New Subgoal Sequences

Given a goal-ordering, SteppingStone attempts to solve the ordered subgoals of the problem by means-ends analysis without undoing them once solved. Although this approach will not usually work for the entire set of subgoals, subsequences of subgoals are often independent and can be solved in this manner. We say that an *impasse* arises if problem solving requires the undoing of some previously solved subgoal. There are two ways of resolving an impasse - by memory or by search.

Memory consists of impasse characterizations and new sequences of subgoals for resolving the impasses. An impasse is characterized by the subgoal being attempted and those solved subgoals that must be undone to solve it. When reaching an impasse, SteppingStone indexes memory by testing whether a characterization in memory is contained within the current context defined by the subgoal being attempted and previously solved subgoals. After successfully indexing memory, SteppingStone is recursively applied to the ordered sequence of subgoals to resolve the impasse.

Figure 1 illustrates an impasse from the blocks world and its resolution. It is a classic example of a non-linear subgoal interaction problem that a linear means-ends problem solver cannot solve. Given the initial and goal states listed, and goal ordering of achieving on(b,c) first, the impasse shown would be encountered. This impasse arises because the final subgoal of on(a,b) cannot be achieved without undoing the previously solved subgoal on(b, c). At the bottom of Figure 1 is a sequence of subgoals for resolving this impasse. This sequence would be stored in memory and indexed by the subgoal being



Impasse Solution: ontable(b), ontable(c), on(b,c), on(a,b)

Figure 1: Blocks World Example of an Impasse

solved, on(a,b), and the subgoal that must be undone, on(b,c).

Note that block d is not mentioned in the impasse solution. This occurs because it is not involved in the impasse. This allows the impasse solution in Figure 1 to work regardless of the number of blocks between block c and block a. This type of generalization is normally very difficult to achieve, but falls out naturally from the use of a subgoal sequence to represent the solution. The impasse characterizations are further generalized by changing domain objects into variables.

2.3 Deriving New Subgoal Sequences

If memory does not contain an entry to resolve an impasse, SteppingStone resorts to search. During the search for the solution to a problem, the irrelevant aspects of the state greatly increase its difficulty. For example, adding more blocks to a blocks world problem makes the problem more difficult even if the blocks are independent of the problem solution. The search procedure used by SteppingStone mitigates this problem by beginning the search within the clearly relevant portions of the problem state. Initially, only those state predicates defining the impasse can be undone during search. If the solution cannot be found in this space, it is opened by allowing additional state predicates to be undone during the search until the search space includes a solution. The predicates chosen are those that are least independent of the ones currently allowed to be undone, as measured by the openness heuristic.

Once a sequence of moves for resolving an impasse is found, it is analyzed to derive a new subgoal sequence. The sequence of moves for resolving the impasse is equivalent to the sequence of states generated by the moves. This sequence of states can be regarded as a sequence of very specific subgoals. Since these subgoals solve an impasse, only those portions of the state involved in the impasse are relevant. This allows the subgoals to be generalized by including in each only those portions of the state involved in the impasse, which is defined by the subgoal being solved and the previously solved subgoals that needed to be undone to solve the impasse. From this sequences repeating subgoals are then removed, yielding the desired sequence of subgoals to be stored in memory.

Problem Size	Subgoal Sequences Learned		Solution Length	
	Number	Length	Found	Optimal
3x3	2	5.0	49	23
4x4	4	6.6	146	53
5x5	6	6.4	306	?
6x6	8	5.9	520	?

Table 1: Features of Solutions and Subgoals Learned

3 Experimental Results

SteppingStone was implemented and applied to the tile sliding domain. Because of its complexity, this puzzle has been a standard benchmark for search-based techniques. For techniques based on goals, the degree of subgoal interaction also makes the problem difficult. Each problem of size NxN, consists of N^2-1 subgoals defining the final location of the tiles. Since the blank was not treated as a subgoal, the problem was not decomposable.

SteppingStone was tested on different sized puzzles ranging from the 3x3 (8-puzzle) to the 6x6 (35-puzzle). The number of reachable problem states in an NxN puzzle is $N^2/2$. For the 3x3 case the number of states is 181440, and for the 6x6 case the number of states is approximately 10^{41}. General problem solvers are rarely applied to this puzzle because of the huge search space. Solutions to random problems from the 6x6 sized puzzle have yet to be published using any system.

We explored Stepping Stone's ability to learn new subgoal sequences, and the effects these sequences had on problem-solving performance. We tested the system's ability to scale to difficult problems. The independence of SteppingStone's learning method from the subgoal ordering, as well as the effects of different subgoal orderings on performance and learning speed were also examined.

3.1 System Performance

Experiment one was designed to test the ability of SteppingStone to learn to improve its performance, and to scale to difficult problems. To do this, it was applied to the tile sliding domains described. Ten orderings of the subgoals were chosen based upon the openness heuristic for each problem size. The system was trained on twenty problems, during which new subgoal sequences were derived and stored in memory. After training, the number and length of the learned subgoal sequences was recorded. The system was then tested on 100 random solvable problems. During testing, learning was turned off, and no new subgoal sequences were added to memory. The length of the solutions found were recorded. Optimal solutions were acquired, when possible, to evaluate the solution quality.[1] Averaged results of these experiments are presented in Table 1.

Somewhat surprisingly, after learning, SteppingStone solved every problem in the test set without resorting

[1]The optimal data for the 3x3 case were compiled using A*. The data and 100 test problems for the 4x4 sized puzzle were taken from Korf [1985a].

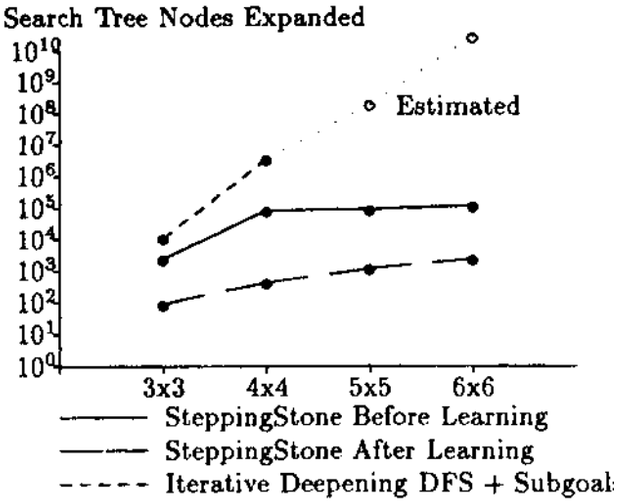


Figure 2: Performance Improvement from Learning

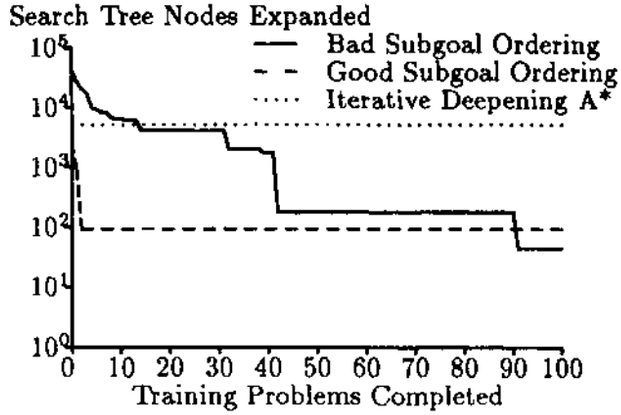


Figure 3: Performance with Different Subgoal Orderings

to search, i.e., it never reached an impasse that it could not overcome with its current small memory of subgoal sequences. Similar results occurred for each of the four different sized puzzles.

SteppingStone learned few new subgoal sequences because it encountered few impasses. This experiment showed that with the orderings derived using the openness heuristic, most subgoals were solved without ever having to undo previously solved subgoals. In addition, when the size of the problem increased, the number of impasses grew at a constant rate, while the size of the problem grew at an exponential rate. Finally, the experiment showed that the solutions derived were about two to three times longer than optimal, which is not surprising since the approach made no attempt at optimization.

Derived subgoal sequences were short, the result of the generalization from domain moves to subgoals. These new subgoals greatly decreased the amount of work required to solve the problems. Figure 2 illustrates the difference in the amount of work required by the system before and after the new subgoal sequences were learned.

To show the SteppingStone approach reduces the amount of work required to solve problems, it is compared to a simple search based approach. Iterative deepening depth-first search was applied to the same ordered sequence of subgoals used by SteppingStone and was tested on the same set of random problems. The work required by this search method to solve the problems is plotted with the SteppingStone data in Figure 2. Because of the huge amount of search required by the search-based approach on the 5x5 and 6x6 sized puzzles, they were conservatively estimated using the growth in the computed branching factor, and assuming an approximate constant growth in the average maximum distance between subgoals. The results are plotted on a log scale to illustrate the exponential decrease in the amount of work required.

The results indicate that learning subgoals significantly reduced the work required for problem solving. This clearly demonstrates the power of the subgoals being learned. The work required before learning was dominated by the search for impasse solutions.

The amount of work required before learning was still much less than that required to exhaustively search for each subgoal solution. This resulted from two factors. First, SteppingStone did not resort to search for every subgoal, but only for those that generated impasses. Second, the openness heuristic was surprisingly effective at curbing the growth in the branching factor by keeping the search for the impasse solution within a small portion of the entire search space.

A classic problem with a learning problem solver is that it can become swamped with its own learning and perform slower, resulting in the problem-solving fan-effect [Minton, 1985]. SteppingStone avoids this problem by learning only when impasses are encountered, and using its knowledge only when needed. Impasses provide sufficient focus to ensure that what is learned is useful, while the indexing assures that what is learned is tried only when it is likely to be useful. This experiment has shown that this type of learning is sufficient to provide a great deal of improvement in problem-solving performance.

3.2 Goal Ordering Effects

SteppingStone was tested in the 3x3 tile sliding domain with four different subgoal orderings to evaluate the ability of the openness heuristic to order the subgoals to both decrease the number of impasses encountered and work required to solve them. The first ordering, a *good* ordering, was derived using the openness heuristic. The second ordering, a *bad* ordering, was derived by ordering the subgoals according to the inverse of the openness heuristic. The third ordering was the standard numeric ordering. The fourth ordering was a randomly generated ordering.

For each subgoal ordering, the system was trained on 100 problems. After solving each problem, the system was tested on a separate set of 20 problems. Figure 3 illustrates the amount of work required to solve the problems in the test set as more problems in the training set are solved when using the good and bad orderings. In addition, the amount of work required by iterative deepening A* using the Manhattan distance metric on the same test set of 20 problems is also plotted. This algo-

Problem Size	Search Tree Nodes Expanded
3x3	145
4x4	897
5x5	3960
6x6	14633

Table 2: Search to Derive Optimal Subgoal Orderings

rithm is a good basis for comparison since it is an optimal knowledge-free, search-based algorithm [Korf, 1985a].

In both cases SteppingStone reduced the amount of work required to solve the problem to less than 100 search tree nodes. As expected, the bad subgoal ordering encountered many more impasses than the good ordering. Using the bad subgoal ordering SteppingStone required thirty new subgoal sequences to solve impasses encountered, while with the good ordering it required only two. Initially, the system using the bad ordering required more work than iterative deepening A*, but after solving fourteen problems required less. The bad ordering required more than an order of magnitude more search to solve the initial training problem than the good ordering. It also required many more training examples to learn the needed new subgoal sequences. Similar results occurred with the linear and random orderings, with the performance curve falling between that for the bad and good ordering. The results imply that the openness heuristic is able to order the subgoals so that fewer impasses are encountered, and the ones encountered are easier to solve.

Interestingly, after learning was completed, the bad ordering of the subgoals required slightly less search to solve the problems than did the good ordering. With the bad ordering, the system almost always had to rely upon the more efficient memory of subgoal sequences. This small amount of improved performance came at the cost of a larger memory, and slower, more costly learning. Nonetheless, as long as solutions to all impasses can be generated, the system performance will always reach a level where the memory of subgoal sequences suffice to solve all encountered impasses, establishing the independence of the learning method from the subgoal orderings.

The good subgoal orderings were derived through a search of the goal space using the openness heuristic. The search began with orderings of length one and then expanded only those nodes corresponding to the best orderings that include an additional subgoal, as measured by the openness heuristic. The amount of search required for each of the domains is recorded in Table 2. The search required was relatively small since there were usually few *best* partial orderings. This search assisted in reducing the number of impasses encountered by deriving a good subgoal ordering. The search was conducted separately for each domain, although using a good representation of the problem offers the opportunity for reuse of subgoal orderings from smaller problems when working on larger ones.

4 Discussion

The results show how learning new subgoal sequences can greatly improve problem-solving performance. By learning only when an impasse was encountered and indexing the knowledge with the impasse context, the utility of the knowledge learned was much higher than its cost to apply. By first ordering the subgoals of the problem with the *openness* heuristic the likelihood of encountering impasses was decreased, along with the amount of learning required. The openness heuristic also proved useful in guiding the search for a solution to the impasse by limiting the portion of the space that needed to be searched. The independence of the learning method from the goal orderings was demonstrated by its success with good, bad, numeric, and random orderings. The generality of the subgoal sequences learned was illustrated by the small number of them required to solve random problems selected from domains with large numbers of difficult problems.

SteppingStone derives its power through the interaction of a number of different methods. First, the ordered subgoals of the problem are initially treated as independent. Independent subgoals greatly decrease the branching factor of a problem [Korf, 1988]. Second, means-ends analysis decreases the search required by focusing on relevant subgoals and operators. Third, learning new subgoal sequences greatly simplifies the difficult portions of the problem by deriving intermediate goals, or stepping stones, to bridge these problem gaps. This, too, has been shown to greatly reduce the difficulty of a problem [Korf, 1988]. Finally, brute-force search provides the power needed to initially solve those difficult problem gaps.

SteppingStone depends upon some regularity in the impasses that are encountered. If it continued to encounter new impasses, it would be forced to continually fall back on costly search. In addition, solutions to the impasses encountered must be accessible. SteppingStone reduces the amount of search required to find impasse solutions by using the openness heuristic to constrain the search.

Finally, since SteppingStone combines two different weak methods, it has opportunities for learning that are unavailable to approaches using a single weak method. SteppingStone takes a sequence of moves generated by a search procedure and translates it back into a goal-based representation that the means-ends problem solver can use.

We intend to test the generality of the approach by applying it to additional domains and by comparing it with other problem solvers, such as Prodigy [Minton, 1988] and SOAR [Laird *et al.*, 1986]. Work has also begun on deriving new subgoal sequences for avoiding impasses rather than solving them once encountered. Finally, comparison on specific problem domains is a weak means for illustrating the power of a problem solver. It is our hope that we can develop more principled techniques for evaluating and comparing learning problem solvers.

Acknowledgements

We would like to thank David Aha, Randy Jones, Patrick Young, and the rest of the UCI Machine Learning group for their comments and discussions concerning this work.

References

[Alterman, 1986] Richard Alterman. An adaptive planner. In Proceedings of the Fifth National Conference on Artificial Intelligence, pages 65-69, Philadelphia, Pennsylvania, 1986.

[Bradtke and Lehnert, 1988] Steven Bradtke and Wendy G. Lehnert. Some experiments with case-based search. In Proceedings of the Seventh National Conference on Artificial Intelligence, pages 133-138, St. Paul, Minnesota, 1988.

[Carbonell, 1986] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Machine Learning: An Artificial Intelligence Approach (Vol 2). Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1986.

[Fikes et ai, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. Artificial Intelligence, 3:251-288, 1972.

[Hammond, 1986] Kristian J. Hammond. Chef: A model of case-based planning. In Proceedings of the Fifth National Conference on Artificial Intelligence, pages 267-271, Philadelphia, Pennsylvania, 1986.

[Iba, 1985] Glenn A. Iba. Learning by discovering macros in puzzle solving. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pages 640-642, Los Angeles, California, 1985.

[Kibler, 1984] Dennis Kibler. Generation of admissible heuristics by problem transformation. Technical Report 225, Irvine: University of California, Department of Information and Computer Science, 1984.

[Kolodner, 1987] Janet L. Kolodner. Extending problem solver capabilities through case-based inference. In Proceedings of the Fourth International Workshop on Machine Learning, pages 167-178, Irvine, California, 1987.

[Korf, 1985a] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27:97-109,1985.

[Korf, 1985b] Richard E. Korf. Iterative-deepening-A*: An optimal admissible tree search. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pages 1034-1036, Los Angeles, California, 1985.

[Korf, 1988] Richard E. Korf. Real-time heuristic search: New results. In Proceedings of the Seventh National Conference on Artificial Intelligence, pages 139-144, St. Paul, Minnesota, 1988.

[Laird et ai, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The anatomy of a general mechanism. Machine Learning, 1(1): 11-46, 1986.

[Langley, 1985] Pat Langley. Learning to search: From weak methods to domain-specific heuristics. Cognitive Science, 9:217-260, 1985.

[Minton, 1985] Steven Minton. Selectively generalizing plans for problem-solving. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pages 596-599, Los Angeles, California, 1985.

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

[Mitchell et ai, 1983] Tom M. Mitchell, Paul E. Utgoff, and Ranan Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In Machine Learning: An Artificial Intelligence Approach. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1983.

[Pearl, 1984] Judea Pearl. On the discovery and generation of certain heuristics. Technical Report UCLA-ENG-CSL-8234, Los Angeles: University of California, Computer Science Department, 1984.

[Porter and Kibler, 1986] Bruce W. Porter and Dennis Kibler. Experimental goal regression: A method for learning problem solving heuristics. Machine Learning, l(3):249-285, 1986.

[Ruby and Kibler, 1988] David Ruby and Dennis Kibler. Exploration of case-based problem solving. In Proceedings of the Case-Based Reasoning Workshop, pages 345-356, Clearwater Beach, Florida, 1988.