Failsafe -- A Floor Planner that Uses EBG to Learn from its Failures

Jack Mostow and Neeraj Bhatnagar*

Rutgers University Computer Science Department
New Brunswick, New Jersey 08903, USA

Abstract

Analysis of failed problem solving efforts enables people to guide subsequent problem solving to avoid similar failures. This kind of learning *while doing* is essential in complex domains. We define Explanation-based Learning from Failure (ELF), a technique for achieving such a capability, and describe its prototype implementation in Failsafe, a Prolog program that learns from failure while solving floor planning problems.

I Introduction

Explanation Based Generalization (EBG) [Mitchell et al 86] has been proposed as a method whereby problem-solvers can speed up with practice by learning from positive examples of problem-solving behavior [Prieditis & Mostow 87]. However, complex domains like design may require an unreasonable amount of brute-force search to find positive examples from which to learn. [Mostow 85] suggests that in such domains the problem solver must do intelligent adaptive search to converge to a solution quickly. One way to do adaptive search is to find the properties that make a generated solution unacceptable and, in future, avoid proposing solutions with these properties. We call such unacceptable solutions *failures*. In this paper, we propose that a system can use EBG to explain its failures in order to get sufficient conditions for them. These sufficient conditions, when negated, give necessary conditions for success which, in turn, can be used as negative heuristics to prune the search.

As a test bed for exploring the approach, we have chosen a simplified floor planning task developed by Prof. Chris Tong for an intermediate AI course. In this domain, a problem consists of fitting rectangular rooms into a rectangular area called 'house' of given integer length and width, subject to the following constraints:

- Room length and width must be integers in the ranges given in the problem.
- Rooms must not overlap.
- Rooms must be inside the house.
- The rooms must cover the entire house area.
- Every room must touch at least one side of the house.
- In addition, a problem may require that a given room be adjacent to another along a specified direction, or to a specified side of the house.

This paper describes Failsafe, a prototype floor planner implemented in Prolog. Failsafe uses Explanation-based Learning from Failures (ELF) to learn new operator preconditions.

II Description of the Technique

We now use an example to introduce ELF. We shall assume that the reader is familiar with the EBG learning paradigm described in [Mitchell et al 86].

Consider a generate-and-test (G&T) problem solver that generates floor plans using the operator 'Locate(R,X,Y,Sx,Sy)' to place room 'R' with width 'Sx', length 'Sy', and front left corner at (X,Y), where the front left corner of the house is at (0,0). An N room floor plan is generated by N applications of this operator. This plan is then passed to the tester module, which tests if all constraints of the problem are satisfied. Initially, operator 'Locate' has no knowledge about which placements of rooms would generate 'obviously wrong' floor plans and should be avoided. It has the following STRIPS-style definition:

<u>Operator</u>: Locate(R,X,Y,Sx,Sy)

<u>Preconditions</u>: room(R) ∧ is_in_house(X,Y) ∧ valid_width(Sx) ∧ valid_length(Sy)

<u>Add List</u>: located(R,X,Y,Sx,Sy)

*This work is supported by NSF under Grant Number DMC-8610507, by the Rutgers Center for Computer Aids to Industrial Productivity, and by DARPA under Contract Number N00014-85-K-0116.

The preconditions check that the object to be placed is a room, that the width and length are in the legal ranges, and that the room's front left corner will fall in the house area.

Suppose the floor planner defined above is given the following trivial floor planning problem: fit two rooms, 'r1' and 'r2', of length 1 and width either 1 or 2, in a house of length 1 and width 3. A solution to this problem is shown in Figure II-1. However, starting from the lower left corner of the house and giving the smallest allowed size to the rooms, the floor planner first generates the plan shown in Figure II-2. Here S_0 is the initial state, S_2 is the final failure state, and the failure path contains two applications of operator 'Locate.'

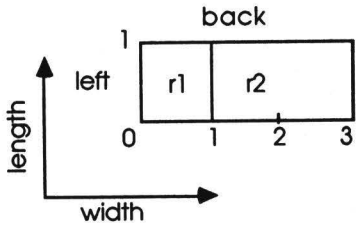

Figure II-1: A Solution to the Example Problem

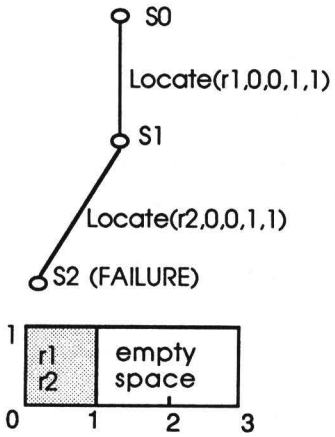

Figure II-2: An Unacceptable Solution

One reason this plan fails is that it has overlapping rooms. The simple G&T floor planner will, at this stage, chronologically backtrack to the point where it made its last choice, say the size of room 'r2', and select the next possible value, namely 2x1. This behavior is doubly stupid, first because the floor planner places the rooms in overlapping positions, and second because in changing the size of 'r2', it tries to repair the faulty plan at a place away from the fault.

Suppose the floor planner has the domain knowledge shown in Figure II-3. Here "failure" is a global assertion which the learning module tries to prove whenever the tester fails a generated solution.

```
failure:- overlap(R1,R2).
overlap(R1,R2) :-   located(R1,X1,Y1,Sx1,Sy1) ∧
                    located(R2,X2,Y2,Sx2,Sy2) ∧
                    ne(R1,R2) ∧
                    northeast(X1,Y1,X2,Y2) ∧
                    extendsbeyond(X1,Sx1,X2) ∧
                    extendsbeyond(Y1,Sy1,Y2).
northeast(X1,Y1,X2,Y2):-  ge(X2,X1) ∧ ge(Y2,Y1).
extendsbeyond(X1,Sx1,X2):- same(X3,X1+Sx1) ∧
                           gt(X3,X2).
```

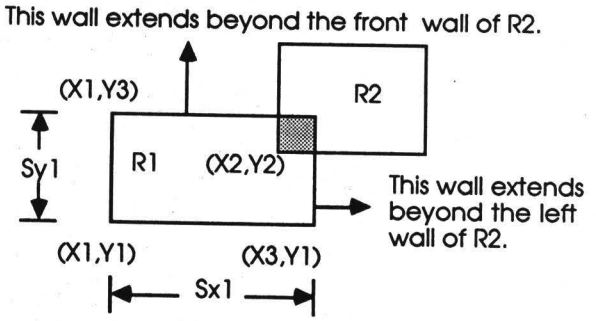

Figure II-3: Domain Theory to Prove Overlap

ELF uses the following four steps to recover and learn from this failure.

(1) Prove the failure: Based on the domain knowledge, explain why the current state S_i is not a goal state and find $C_i = F_1 \wedge F_2 \wedge \ldots \wedge F_k$, which is a sufficient but specific condition for failure. Here, each conjunct F is considered operational, i.e., verifiable without search, whether by database lookup (`located`) or procedure (`gt`).

In our example, we get

C_2= located(r1,0,0,1,1) ∧
located(r2,0,0,1,1) ∧ ne(r1,r2) ∧ ge(0,0) ∧
ge(0,0) ∧ same(1,0+1) ∧ gt(1,0) ∧
same(1,0+1) ∧ gt(1,0).

(2) Get a general condition 'G' for failure: Regress C_i through the proof tree to get the generalization, G_i, of C_i such that the same proof of failure holds.

In our example, C_2 can be generalized to G_2, given below.

G_2 = located(R1,X1,Y1,Sx1,Sy1) ∧
located(R2,X2,Y2,Sx2,Sy2) ∧ ne(R1,R2) ∧
ge(X2,X1) ∧ ge(Y2,Y1) ∧
same(X3,X1+Sx1) ∧ gt(X3,X2) ∧
same(Y3,Y1+Sy1) ∧ gt(Y3,Y2).

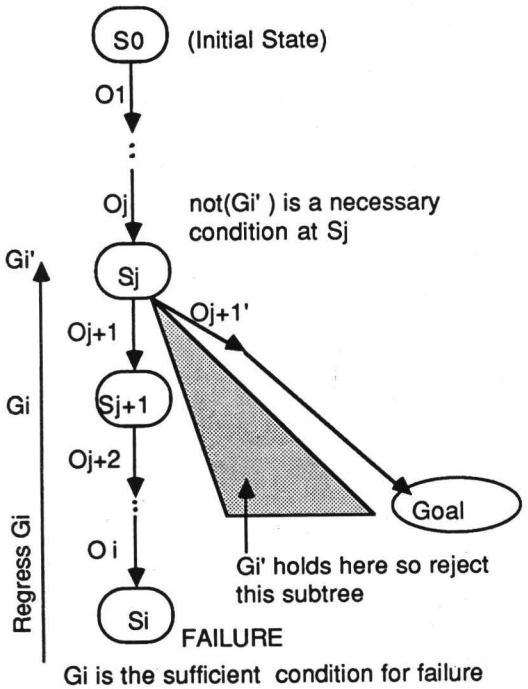

Figure II-4: Regress the failure condition back.

(3) Regress 'G' up the failed path to the place of recovery: Suppose that state S_i was reached through states S_1, ..., S_j, ..., S_{i-1} by applying operators (not necessarily distinct) O_1 through O_i to the initial state S_0. Regress G_i back through the operators O_i, O_{i-1}, ..., O_{j+1}, until it changes to some different condition $G'_i = \text{Regress}(G_i, O_{j+1}) \neq G_i = \text{Regress}(G_i, O_{j+2} \ldots O_i)$, as shown in Figure II-4.

If G'_i holds in a state S_j, the operator sequence O_{j+1}, ..., O_i will *not* take the solver to the goal state. In fact, Failsafe assumes that all instantiations of O_{j+1} which satisfy G'_i will lead to failure, no matter which operators are applied subsequently. This assumption is justified in Failsafe because it has no operators besides Locate. In general, justifying this assumption would require proving that no other operators applicable at each point in the path can lead to success. Such proofs require domain knowledge or exhaustive search (as in PRODIGY [Minton & Carbonell 87]) in amounts unrealistic for adaptive search in complex domains. Fortunately, this requirement is stronger than necessary: safety requires only that ELF not prevent the problem-solver from reaching a correct solution. Section V.A discusses this issue in more depth.

Notice that the condition being regressed back may pass unchanged through one or more operator applications, i.e., $\text{Regress}(G_i, O_{irrelevant}) = G_i$. In learning from this failure, we can simply filter out such operators from the operator sequence, since they have no effect on the sufficient condition for failure. An interesting case occurs when all but one of the operators are thrown out. In this case we discover a new precondition of the single operator left in the sequence after throwing irrelevant operators out.

In our example, the regression process yields:

G_2' = room(R2) ∧ is_in_house(X2,Y2) ∧ valid_width(Sx2) ∧ valid_length(Sy2) ∧ located(R1,X1,Y1,Sx1,Sy1) ∧ ne(R1,R2) ∧ ge(X2,X1) ∧ ge(Y2,Y1) ∧ same(X3,X1+Sx1) ∧ gt(X3,X2) ∧ same(Y3,Y1+Sy1) ∧ gt(Y3,Y2).

This condition describes a situation in which a room 'R2' to be placed with width 'Sx2,' length 'Sy2,' and lower-left corner at location (X2,Y2) would overlap with an already located room 'R1'. G_2' can be treated as a negative heuristic: when placing 'r2' the planner should make sure that G_2' not hold. Since G_2' has no problem-specific constant, we add $\neg G_2'$ to the preconditions for the operator 'Locate', as shown below. The resulting conjunction could be simplified, but Failsafe does not do so.

<u>Operator</u>: Locate(R,X,Y,Sx,Sy)

<u>Preconditions</u>: room(R) ∧ is_in_house(X,Y) ∧ valid_width(Sx) ∧ valid_length(Sy) ∧ ¬ [room(R) ∧ is_in_house(X,Y) ∧ valid_width(Sx) ∧ valid_length(Sy) ∧ Located(R1,X1,Y1,Sx1,Sy1) ∧ ne(R,R1) ∧ ge(X,X1) ∧ ge(Y,Y1) ∧ same(X3,X1+Sx1) ∧ gt(X3,X) ∧ same(Y3,Y1+Sy1) ∧ gt(Y3,Y)]

<u>Add List</u>: Located(R,X,Y,Sx,Sy)

(4) Resume the search: Select a new instantiation O'_{j+1} of O_{j+1} for which G'_i fails to hold, and continue the search until the goal state is reached or another failure is detected. If no such instantiation exists and no other operator is applicable, regress G'_i further up until it undergoes another change at operator O_m, $m<j+1$, and repeat. If, backtracking in this fashion, the search reaches state S_0, and no instantiation of O_1 is applicable, and no other untried operator applies at S_0, then the problem does not have a solution.

In our example the problem solver will now try alternate sizes and locations for room 'r2' that do not overlap the room 'r1'.

III Description of the Architecture

In designing Failsafe, we chose a very simple generate and test architecture, because it provides failures in abundance and is easy to implement.

Failsafe's **solution generator** initially uses exhaustive search to place the rooms, starting from the front left corner of the house space and going left to right and front to back. After every application of the operator 'Locate(R,X,Y,Sx,Sy),' the generator adds the new fact 'located(R,X,Y,Sx,Sy)' to the problem state. When all rooms are located, the solution is passed to the tester.

The **tester** checks if the generated floor plan meets all the problem constraints. If not, it rejects the solution.

The **learning module** uses ELF to learn new preconditions for the operator 'Locate'. It has a knowledge base of theorems about floor planning, some of which are shown in Figure II-3. Whenever the tester rejects a floor plan, Failsafe uses these theorems to try to prove the goal "failure" in the context of the failed plan. Failsafe explains and generalizes the failure using a Prolog implementation of EBG [Kedar-Cabelli & McCarty 87].

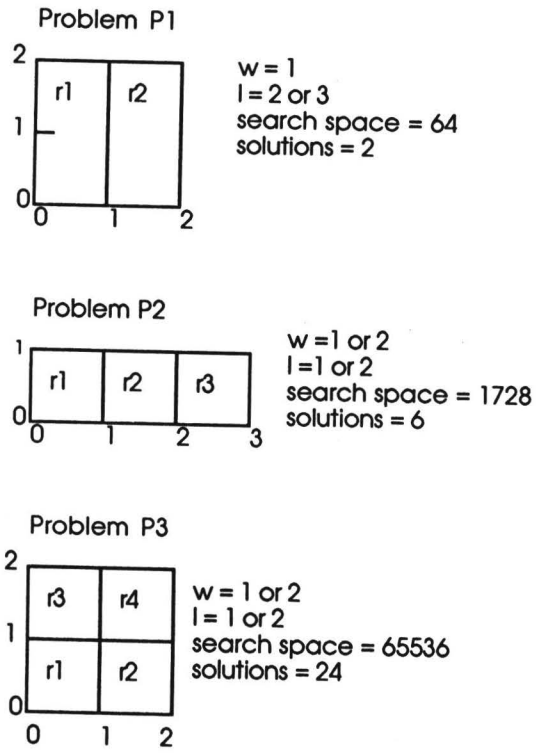

Figure IV-1: Three Example Problems

IV Results of the Implementation

We now present the results of experiments carried out to measure Failsafe's performance, the effect of increasing problem complexity, and transfer of learning from one problem to another.

A Performance on Three Problems

To test the effectiveness of ELF, we ran Failsafe on three example problems of increasing complexity, shown in Figure IV-1.

We measured Failsafe's performance by counting the number of PROLOG subgoals generated while solving each problem. We ran Failsafe on these problems with learning switched off, while learning, and after learning. Table IV-1 shows the results.

Table IV-1: Failsafe's performance data

Problem	Search space	Solutions	Without learning	During learning	After learning
P1	64	2	1061	10124	499
P2	1728	6	120656	53031	3163
P3	65536	24	150000	108830	15348

Improvement in performance is apparent for each of the three problems. Even for small problems the learning overhead is outweighed by improvement in performance, thanks in large part to the inefficiency of Failsafe's initial G&T problem solver. Performance after learning is, as expected, better than during learning, although the difference is not as dramatic as in systems that learn from positive examples. For instance, Soar [Laird et al 86] learns a chunk for an entire problem, which enables it to solve the same problem in a single step the next time around.

B Between-problem Transfer

Table IV-2 shows the between-problem transfer effect for each pair of example problems. We measured this effect by training Failsafe on the first problem and then testing it on the second.

Table IV-2: Transfer Effect

Problem	After learning P1	After learning P2	After learning P3
P1	499	559	559
P2	31525	3163	3163
P3	72846	15348	15348

Solving P2 or P3 teaches Failsafe not to overlap rooms and not to cross house boundaries. Transfer is therefore ideal from P2 to P3, i.e., P2 trains Failsafe for P3 as well as P3 itself would.

Since no boundary-crossing happens to occur during the normal solution of P1, transfer is negative from P2 or P3 to R1: checking for crossed boundaries only slows things down. Since accumulating control knowledge indiscriminately can degrade performance, it may be a good idea to store and test only those preconditions that are frequently violated.

V Open Issues

Currently Failsafe finesses several issues.

A Safety

ELF is defined as *safe* whenever it does not prevent the problem-solver from finding a correct solution. ELF uses a single failed path to reject an entire subtree without searching it exhaustively. While this strategy is crucial for efficient adaptive search of large spaces, it risks losing the ability to find a solution. ELF's safety depends on the problem-solver's operators and the nature of the failed constraint.

A constraint is *monotonically necessary* if, once violated in a partial plan, it will remain violated in all extensions of this plan [Mostow 83a]. Such constraints are safe to learn as preconditions, since no solutions are thereby rendered unreachable; we will call violations of them *monotonic failures*. For example, room overlap is a monotonic failure in Failsafe, since it cannot be repaired by placing additional rooms.

If the effects of an operator can be reversed by other operators, there can be no monotonic failures, since every partial plan can be extended into a (possibly non-optimal) plan that leads to the goal state, assuming one exists. Failsafe's single operator, 'Locate,' has no such inverse.

Even if a condition is not monotonically necessary, it can be treated as such provided there are ways to solve the problem without violating it. Such a condition can be called *pseudo-monotonically necessary*. For example, suppose Failsafe had an operator for sliding an already-placed room from one location to another. Non-overlap would be pseudo-monotonically necessary if every room could be placed without sliding.

Some conditions are not even pseudo-monotonically necessary. For instance, the constraint that every point in the house be covered by some room can be made true by placing additional rooms. Even such constraints can safely be learned as operator preconditions by restricting the circumstances under which they are tested. For example, if the covering condition is only applied to complete floor plans, we can learn that the last room to be placed must include the remaining uncovered points.

B The Hopeless State Problem

At an intermediate state in planning, decisions made during the placement of earlier rooms can make it impossible to satisfy the preconditions of 'Locate' for the next room. We call such a state *hopeless*. An example of a hopeless state is shown in Figure V-1. Here room 'r2' is required to be adjacent to the left and front walls of the house, but room 'r1' has already been placed there, so 'r2' cannot be placed as required.

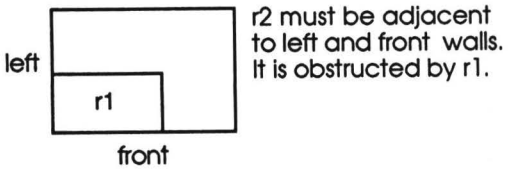

Figure V-1: A Hopeless State

Learning to efficiently recognize hopeless states might enable Failsafe to handle much larger problems. Currently Failsafe recognizes hopeless states only after exhaustive search fails to satisfy the preconditions of an operator. We plan to extend Failsafe to invoke learning at this point; at present learning occurs only when a complete floor plan is rejected by the tester.

C The Non-operational Precondition Problem

To be *operational* in the sense defined in [Mostow 81, Mostow 83b], an operator precondition must refer only to information available when the operator is selected, such as the operator's parameters and the steps used to get to the current state. The non-operational precondition problem occurs when the learned precondition refers to parameters bound by operators selected later on. For example, when the non-overlap condition is regressed back through the sequence locate(r1,X1,Y1,W1,L1);locate(r2,X2,Y2,W2,L2), the resulting condition refers not only to X1, Y1, W1 and L1 but also to X2, Y2, W2 and L2. In fact, the regressed condition is a precondition for the two-operator macro, not for the first operator by itself.

One possible approach is to constrain X1, Y1, W1, and L1 to leave at least one acceptable assignment for X2, Y2, W2, and L2. However, evaluating this condition involves searching for such an assignment, and is therefore liable to defeat the purpose of testing preconditions -- enhancing efficiency.

D Detecting Failures

To learn from its failures, a planner must first detect them. In general, detection of failures is difficult. Failsafe gets around this difficulty because in its domain the length of the target plan is fixed and known: exactly N applications of the operator

'Locate' are required to complete an N room floorplan. Failure detection requires other methods when the maximum length of the solution path is not known, such as noticing cycles [Langley 83, Minton & Carbonell 87].

E Which Necessary Conditions to Test

Explanation-based learning becomes counter-productive when so many macro operators are learned that testing their preconditions costs more than rediscovering them by search [Minton 85]. ELF is potentially subject to the same problem.

A system like Failsafe could sample how often a given condition fails and, before trying an operator, test only those which fail more frequently than some threshold that reflects the relative costs of precondition testing and search. This threshold could be modified dynamically depending on the performance of the system.

VI Related Work

The usefulness of negative heuristics learned by analyzing problem-solving failures has been reported previously. [Kibler & Morris 83] reports four negative heuristics obtained by manually analyzing "stupid" actions taken by a blocks world planner. By guiding the planner to avoid such actions, these four heuristics reduce search to almost zero.

CHEF [Hammond 86] uses domain knowledge to anticipate the failures that might occur while trying to satisfy a goal. When such failures occur, this knowledge is used to repair the plan instead of re-planning from scratch.

[Siklossy & Dowson 77] describes preprocessing of STRIPS operators to discover conditions that cannot simultaneously hold in any legal state. Such conditions become indicators of failures. For example, inspection of the operator definitions reveals that no state can satisfy holding(X) and clear(X) for the same block X, since operators that assert one delete the other. Knowledge gained by preprocessing the operators is used to prune planning states that satisfy two or more mutually inconsistent conditions.

FOO [Mostow 83a] refines a G&T search procedure into a heuristic search by using monotonically necessary conditions to prune partial paths, and by using monotonically sufficient conditions to reorder the search. While FOO uses static analysis to refine the procedure without ever executing it, Failsafe relies on executing the search to identify the conditions that actually lead to failure in practice. FOO uses domain knowledge to split tests into monotonic components so it can move them, while Failsafe lacks the ability to reformulate its tests. FOO mechanically applies the sequence of program transformations that modifies the search procedure, but relies on a user to select the sequence. Unlike FOO, Failsafe is completely automatic.

PRODIGY learns search control knowledge from the failure of an entire subtree to produce a solution [Minton & Carbonell 87]. In contrast, ELF learns from a single failed path, though the resulting control knowledge is over-general for non-monotonic failures.

ELF's technique for backtracking to the closest node where the failure might be repaired resembles the dependency directed backtracking method used in the EL system [Stallman 77]. However, unlike Failsafe, EL does not generalize.

VII Conclusion

We have presented an implemented technique for learning useful search control knowledge from failures that occur during search. Failsafe's performance improvement more than compensates for its learning overhead, even for small problems, and even when learning *while doing*.

We have shown how EBG can be used to learn from negative examples, not just positive ones. **Learning from failure while problem-solving makes it possible to solve problems too complex to solve otherwise.** However, learning from success seems to create more powerful control knowledge, such as Soar's chunks for achieving goals in one step. Future research should compare and integrate methods for learning from success and failure.

Acknowledgments

We thank Chris Tong and Prasad Tadepalli for their excellent ideas during various stages of this work, Smadar Kedar-Cabelli for her Prolog implementation of EBG, Lou Steinberg for his continuing encouragement, and Steve Minton and an anonymous referee for their helpful comments.

References

[Hammond 86] Kristian J. Hammond.
Learning to anticipate and avoid planning problems through the explanation of failures.
In *AAAI86*, pages 556-560. American Association for Artificial Intelligence, Philadelphia, PA, 1986.

[Kedar-Cabelli & McCarty 87]
Kedar-Cabelli, S. T. and McCarty, L. T.
Explanation-Based Generalization as Resolution Theorem Proving.
In *Proceedings of the Fourth International Machine Learning Workshop*. Morgan Kaufmann, University of California at Irvine, June, 1987.

[Kibler & Morris 83]
Dennis Kibler and Paul Morris.
Don't be Stupid.
In *IJCAI83*, pages 345-347. Karlsruhe, Germany, 1983.

[Laird et al 86] J. E. Laird, P. S. Rosenbloom, and A. Newell.
Chunking in Soar: the anatomy of a general learning mechanism.
Machine Learning 1(1):11-46, 1986.

[Langley 83] Pat Langley.
Learning effective search heuristics.
In *IJCAI83*, pages 419-421. Karlsruhe, Germany, 1983.

[Minton 85] Steven Minton.
Selectively generalizing plans for problem-solving.
In *Proceedings IJCAI85*, pages 596-599. Los Angeles, CA., August, 1985.

[Minton & Carbonell 87]
S. Minton and J. G. Carbonell.
Strategies for learning search control rules: an explanation-based approach.
In *Proceedings IJCAI87*. Milan, Italy, August, 1987.

[Mitchell et al 86]
T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli.
Explanation-based generalization: a unifying view.
Machine Learning 1(1):47-80, 1986.

[Mostow 81] J. Mostow.
Mechanical Transformation of Task Heuristics into Operational Procedures.
PhD thesis, Carnegie-Mellon University, 1981.
Technical Report CMU-CS-81-113.

[Mostow 83a] D. J. Mostow.
Learning by being told: machine transformation of advice into a heuristic search procedure.
In J. G. Carbonell, R. S. Michalski, and T. M. Mitchell (editors), *Machine Learning*. Palo Alto, CA: Tioga Publishing Company, 1983.

[Mostow 83b] J. Mostow.
A problem-solver for making advice operational.
In *AAAI83*, pages 279-283. American Association for Artificial Intelligence, Washington, DC, August, 1983.

[Mostow 85] J. Mostow.
Toward better models of the design process.
AI Magazine 6(1):44-57, Spring, 1985.

[Prieditis & Mostow 87]
A. Prieditis and J. Mostow.
PROLEARN: Towards A Prolog Interpreter that Learns.
In *Proceedings AAAI87*. Seattle, WA, July, 1987.

[Siklossy & Dowson 77]
L. Siklossy and Clive Dowson.
The role of preprocessing in problem solving systems.
In *IJCAI-5*, pages 465-471. Cambridge, MA, 1977.

[Stallman 77] Stallman, R., and Sussman, G.
Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis.
Artificial Intelligence 9:135-196, 1977.