Strategies for Learning Search Control Rules: An Explanation-based Approach

Steven Minton and Jaime G. Carbonell

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Previous work in explanation-based learning has primarily focused on developing problem solvers that learn by observing solutions. However, learning from solutions is only one strategy for improving performance. This paper describes how the PRODIGY system uses *explanation-based specialization* to learn from a variety of phenomena, including solutions, failures, and goal-interactions. Explicit *target concepts* describe these phenomena, and each target concept is associated with a strategy for dynamically improving the performance of the problem solver. Explanations are formulated using a theory describing the domain and the PRODIGY problem solver. Both the target concepts and the theory are declaratively specified and extensible.[1]

1. Introduction

Recent research has demonstrated that explanation-based learning (EBL) is a viable method for acquiring search control knowledge [22,6,18,27,29]. Almost all EBL problem solvers learn by analyzing why a solution succeeds in solving a problem. The result is a knowledge structure (such as a macro-operator or schema) that describes how to solve similar problems should they arise in the future.

However, explanation-based learning can also be used to learn why a problem-solving method failed. By explaining why a method failed to solve a problem or sub-problem, a system can learn when to avoid that method in the future [16,11,24). In fact, EBL is a very general technique, and in principle an EBL system can learn from any phenomenon that it can explain. To do so, however, a system must know what phenomena - or *target concepts* - to explain and be able to formulate appropriate explanations [12]. Furthermore, the system must be able to modify its behavior appropriately on the basis of the learned information.

This paper describes how the PRODIGY system learns from multiple target concepts in order to increase the utility of the learning process. Specifically, unlike previous EBL problem-solvers, PRODIGY learns from a variety of phenomena, including success, failure, and goal-interactions. Each of these meta-level target concepts corresponds to a strategy for optimizing PRODIGY'S effectiveness as a problem solver. Explanations are formulated using an explicit theory describing the domain and the PRODIGY problem solver. Both the set of target concepts and the theory are declarative and extensible, therefore additional optimization strategies can be incorporated into the system when necessary. This paper also introduces an new EBL algorithm, Explanation-based Specialization (EBS), which constructs explanations by specializing proof schemas using information gained from a training example. EBS is particularly appropriate for problem solvers that learn from their own problem solving performance.

[1] *This* research was sponsored In part by the Defense Advanced Research Prefects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520, in part by the Office of Naval Research under Contract N00014-84-K-0345. and in part by a gift from the Hughes Corporation. In addition, the first author was supported by a Bell Laboratories Ph.d Scholarship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Air Force Office of Scientific Research or the US government.

[2] Goal concept is an equivalent term, but is easily confused with the goal of the problem-solver.

[1] The use of incomplete, approximate, and default theories are topics of current research, but not considered here.

2. The Terminology of Explanation-based Learning

Several recent papers [22,6,*17]* have contributed greatly to the emergence of a relatively standard terminology for describing explanation-based learning. (We use the generic term "explanation-based learning" to refer to deductive learning from single examples, as exemplified by *[17,22,*23,9,8].)

The *target concept* is a high-level description of a concept (i.e., a class of instances).[2] A *training example* is an instance of the target concept. Using the *domain theory,* a set of axioms describing the domain, one can explain why the training example is an instance of the target concept. The explanation constitutes a proof that the training example satisfies the target concept[3]. By finding the weakest conditions under which the explanation holds, EBL will produce a *learned description* that is both a generalization of the training example, and a specialization of the target concept. Typically, the purpose of the learning process is to produce a description that can serve as an efficient recognizer for the target concept.

As an example (adapted from [221) consider the target concept (SAFE-TO-STACK *x y),* that is, object *x* is can be safely placed on object y without object y collapsing. A training example could be a demonstration that a particular book, 'Principles of AI", can be safely placed upon a particular table, Coffee-Table-1. Let us suppose that our domain theory, shown below, contains assertions enabling us to prove that "Principles of AI" is safe to stack on Coffee-Table-1 because all books are lighter than tables. From this proof we can conclude that that any book can be safely stacked on any table. Thus the learned description is (AND (IS-BOOK *x)* (IS-TABLE y)).

DOMAIN THEORY.
(IS-BOOK PRINCIPLESOF-AI)
(SAFE-TO-STACK x y) If (OR (LIGHTER x y) (NOT-FRAGILE y))
(LESS-THAN *w* 5-LBS) if (AND (IS-BOOK *x)* (WEIGHT *x w))*

3. The PRODIGY System

PRODIGY is a learning apprentice [20] that acquires problem solving expertise by interacting with an expert, by carrying out experiments, and, as described in this paper, by analyzing problem solving traces, PRODIGY can be divided into four subsystems:

- An advanced Strips-like[9] problem solver that provides a uniform control structure for searching with both inference rules and operators. The problem solver includes a simple reason-maintenance system, allows operators with conditional effects to be specified, and is capable of interleaving goals. The problem solver's search can be guided by domain-independent or domain-specific control rules.

- An explanation-based learning facility [19,15] that can propose explanations about why a control decision was appropriate, and transform them into search control rules.
- A learning-by-experimentation module [4] for refining incompletely or incorrectly specified domain knowledge. Experimentation is triggered when plan execution monitoring detects a divergence between internal expectations and external expectations.
- A user-interface that can participate in an apprentice-like dialogue, enabling the user to evaluate and modify the system's behavior.

Except for the experimentation module (which is currently under development), PRODIGY has been fully implemented, and tested on several task domains including a machine shop scheduling domain and a 3-D robotics construction domain. For brevity of exposition the examples described in this paper will be taken from an extremely simple task domain, the familiar "blocks world", whose standard specification [251 is shown in Figure 3-1. While the blocks world may seem contrived, solving apparently simple problems may involve searching hundreds of nodes. In other words, the domain is simple but as we will see, the formalization (i.e., axiomization) of the domain is far from ideal. By acquiring search control knowledge, PRODIGY effectively modifies its behavior so that its performance reflects the inherent simplicity of the domain, rather than the deficiencies of the formalization. Learning search control knowledge thus has the same effect as reformalization ~ important knowledge is brought to bear early, so that correct decisions can be made efficiently.

PICKUP *b*
Precondition: (AND (CLEAR *b)* (ONTABLE *b)* (ARMEMPTY))
Add: (HOLDING *b)*
Delete: (ONTABLE *b)* (CLEAR *b)* (ARMEMPTY)

PUTDOWNb
Precondition: (HOLDING *b)*
Add: (ONTABLE *b)* (CLEAR *b)* (ARMEMPTY)
Delete: (HOLDING *b)*

STACK b1 b2
Precondition: (AND (HOLDING *b1)* (CLEAR b2))
Add: (ON *bl bl)* (CLEAR *bl)* (ARMEMPTY)
Delete: (HOLDING *bl)* (CLEAR *b2)*

UNSTACK *bl bl*
Precondition: (AND (ON *bl bl)* (CLEAR *bl)* (ARMEMPTY))
Add: (HOLDING M) (CLEAR *bl)*
Delete: (ON *bl bl)* (CLEAR M) (ARMEMPTY)

Figure 3-1: Blocks World Specification

3.1. An Example Problem

A problem consists of a description of a start-state, and a set of goal states, as shown in Figure 3-2. PRODIGY'S description language, called PDL, is a logic-based language that includes conjunction, disjunction, negation, existential quantification, and universal quantification over sets. (Variables are shown in lower case.) PDL is also used for describing the preconditions of operators and search control rules.

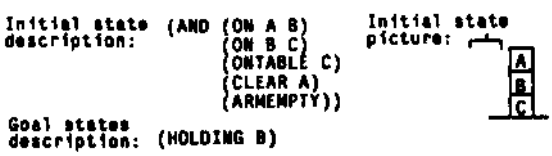

Figure 3-2: An Example Problem

By default, PRODIGY searches depth-first, attacking goals from left to right, unless search control rules indicates a more appropriate ordering. To solve a goal, the system selects an operator with a postcondition (a formula in the add or delete list) that matches the goal. If the preconditions of the operator are satisfied, then the operator can be applied, otherwise the system subgoals on any unsatisfied preconditions. (For simplicity, this paper describes only the basic capabilities of the system necessary to illustrate the examples. More sophisticated features, such as conditional effects, inference rules, functions, and non-linear planning are discussed in [15]. Readers unfamiliar with the basic operation of a STRIPS-style system should see [251.)

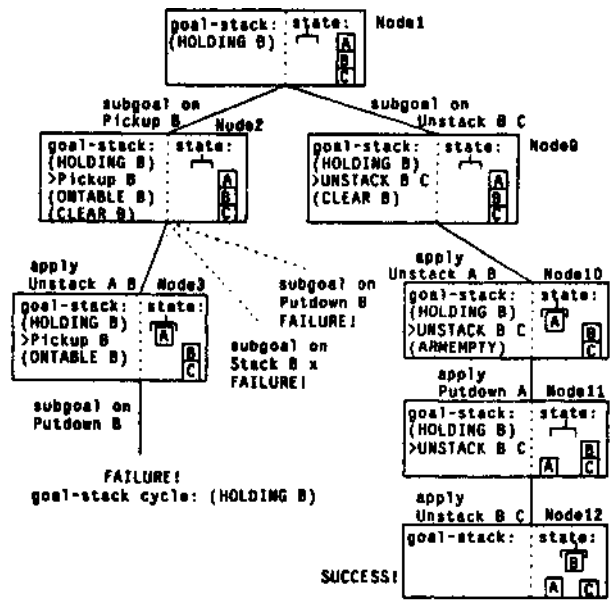

Figure 3-3: Search Tree

To solve the example problem, PRODIGY considers applying either PICKUP or UNSTACK to achieve (HOLDING B), as illustrated in Figure 3-3. PICKUP is inappropriate, but this is only discovered during the course of solving the problem. In attempting to PICKUP B, PRODIGY is forced to subgoal on (CLEAR B) and (ONTABLE B), the unsatisfied preconditions of PICKUP. These are added to the goal-stack, and then PRODIGY attempts to solve (CLEAR B). Unstacking A from B achieves (CLEAR B), but then a goal-stack cycle occurs in attempting to achieve (ONTABLE B). PRODIGY then attempts other ways to (CLEAR B) at Node2, but these also result in failure. (The problem solver has no way of knowing that unstacking A from B did not cause the failure of Node3. But as we will see, when learning is interleaved with problem-solving, PRODIGY backtracks more intelligently.) Eventually backtracking to Node1, PRODIGY attempts unstacking B from C, which leads to a solution.

PRODIGY'S performance in a given domain can be improved by the addition of search control rules. Control rules constrain the search by mediating the four decisions PRODIGY makes during each problem solving cycle. First, the system must choose a node in the search tree to expand next - the default is depth-first expansion. Each node consists of a goal-stack, a set of subgoals and a state describing the world. Once a node has been chosen, one of the subgoals must be selected, and then an operator relevant to reducing this subgoal. Finally, a set of bindings for the variables in that operator must be decided upon.

Each of these four decisions can be affected by control rules, which indicate when to SELECT, REJECT, or PREFER a particular candidate (node, goal, operator or bindings). Given a default set of

candidates, PRODIGY first applies the applicable selection rules to select a subset of the candidates. If no selection rules are applicable, all the candidates are selected. Then rejection rules are executed to filter this set, and finally preference rules are used to find the heuristically best alternative . (If backtracking is necessary, the next most preferred is attempted, and so on.)

Consider the first control rule shown in Figure 3-4. This rule is a operator selection rule; it is used in deciding which candidate operator to choose. Notice that the description language PDL allows the use of meta-level predicates such as KNOWN. The formula (KNOWN *node exp)* is true if the expression *exp* matches the state at *node.* Therefore the rule asserts that if the current goal matches (HOLDING *x)* and *x* is not on the table, then UNSTACK is the appropriate operator. Normally, since both PICKUP and UNSTACK can achieve (HOLDING *x),* both would be candidate operators. If this rule had been learned prior to our example problem (Figure 3-3), PRODIGY would have solved the problem without having to waste time exploring Node2 and its descendents.

```
(SELECT OPERATOR (UNSTACK x y))
if (and (CURRENT-NODE node)
        (CURRENT-GOAL node (HOLDING x))
        (CANDIDATE-OPERATOR node (UNSTACK X y) )
        (KNOWN node (NOT (ONTABLE X)) ) ) )

(PREFER GOAL (ON X y) OVER (ON w X))
if (and (CURRENT-NODE node)
        (CANDIDATE-GOAL node (ON X y) )
        (CANDIDATE-GOAL node (ON w X) ) )
```

Figure 3-4: Two Search Control Rules

The second control rule in Figure 3-4, a goal preference rule, is used to decide which of the candidate goals at a node to attempt first. The rule states that if (ON *w x)* and (ON *x y)* are candidate goals, then it is preferred that (ON *x y)* should be attempted before (ON *w x).* This piece of control knowledge directs the problem solver to build stacks of blocks from the bottom up. Since PRODIGY finds the *most* preferred goal from a set of candidates, this rule correctly handles stacks regardless of height (i.e., the number of goals in the ON chain).

4. Learning Control Rules: The Explanation-based Specialization Method

PRODIGY'S explanation-based learning module can either be invoked after the problem solver has finished, or learning can be interleaved with problem solving. In either case, the learning process begins by examining the a trace of the explored search tree (in a pre-order traversal) in order to pick out examples of PRODIGY'S target concepts. Search control rules are learned by explaining why a training example satisfies a target concept. There are currently four types of target concepts implemented in PRODIGY:

1. SUCCEEDS: A control choice succeeds if it results in a solution. Learning about successes results in preference rules.
2. FAILS: A choice fails if there is no solution consistent with that choice. Learning about failures results in rejection rules.
3. SOLE-ALTERNATIVE: A choice is a sole-alternative if all other candidates fail. Learning about sole alternatives results in selection rules.
4. GOAL-INTERACTION: A choice causes in a goal-interaction if it results in a situation where a goal must be re-achieved. Learning about goal-interactions results in preference rules.

Each type of target concept has four variants, one for each of the four control decisions (picking a node, goal, operator, and bindings). For example, the target concept (OPERATOR-FAILS *op g n)* is true if operator *op* fails to solve goal *g* at node n. As with all concepts in the system, target concepts are represented by atomic formulas. Each target concept is declaratively specified to the system, as illustrated by Figure 4-1. A target concept specification includes a template for building search control rules, and may include other information such as heuristics for selecting training examples, as described in [151.

```
Target Concept: (OPERATOR-FAILS op goal node)
Rule Template:  (REJECT OPERATOR op)
                    if (AND (CURRENT-NODE node)
                            (CURRENT-GOAL node goal)
                            (CANDIDATE-OPERATOR Op node)
                            (OPERATOR-FAILS op goal node))
```

Figure 4-1: Target Concept Specification for OPERATOR-FAILS

An explanation describes why an example is a valid instance of a target concept, and corresponds to a proof. To construct such proofs, we require a theory describing the problem-solver, as well as a theory that describes the task domain (such as the blocks world). Therefore, PRODIGY contains a set of *architecture-level* axioms which serve as its theory of the problem solver. The domain theory is given by a set of *domain-level* axioms extracted from the domain operators by a pre-processor. Together, these sets of axioms are referred to as *proof-schemas.* Each proof-schema is a conditional which describes when a concept is true. Appendix 1.1 illustrates the architecture-level proof schemas relevant to the concept OPERATOR-SUCCEEDS. These schemas state that an operator succeeds in solving a goal at a node if:

- the operator is applicable and directly solves the goal, or
- subgoaling occurs, creating a child node where the operator succeeds, or
- another operator is applied to achieve a subgoal, creating a child node where the operator succeeds.

Domain-level schemas (Appendix 1.2) indicate the available operators, and their effects and preconditions.

To prove that a target concept is satisfied by an example, PRODIGY specializes the target concept in accordance with the example. This process is equivalent to creating a proof tree by starting at the root (i.e., the target concept) and incrementally expanding the leaves of the tree. To specialize a concept PRODIGY retrieves a proof schema that implies the concept and recursively specializes all the subconcepts in the schema, as described in Figure 4-2. The process terminates when primitive concepts are encountered. The result of the specialization process is a description of the weakest premises of the explanation.

If a concept is described by more than one proof schema, as is OPERATOR-SUCCEEDS, then it is *disjunctive.* When specializing disjunctive concepts, PRODIGY uses the example to determine which schema gives the appropriate specialization[5] Specifically, we allow each concept to be associated with a *discriminator* function which examines the problem solving trace and selects a schema consistent with the training example. In this way, discriminator functions provide a mapping between the example and the explanation.

After the EBS process terminates, the resulting description is a specialization of the target concept. A search control rule is formed by retrieving the rule construction template (found in the target concept specification) and substituting the specialized description

[4]Preferences are transitive. If there is a cycle in the preference graph, then those preferences in the cycle are disregarded. A candidate is best, or "most preferred", if there is no candidate that is preferred over it.

[5]There can be multiple training examples for a subconcept if the subconcept is within the scope of a universal quantifier. Universally quantified statements take the form FORALL *x* SUCH-THAT (P x), (Q x). The formula (P x) acts as a generator for values of x. (Q x) can be any expression. The implementation treats universally quantified statements as if they were written in the logically equivalent normal form: FORALL *x* (OR (NOT (P x))(Q *x))).* Because (P x) is negated, it is not specialized. (However, if P generates a fixed set of values, it can typically be simplified out of the learned description using simplification axioms! [15].) EBS specializes Q separately for each example of x, and returns the disjunction of these descriptions as the specialization of Q

A concept it represented by an atomic formula.
To specialize a concept:

- If the concept is primitive - there is no schema that implies the concept - then return the concept unchanged, otherwise,
- Call the discriminator function associated with the concept to retrieve a schema consistent with the training example. Each non-negated atomic formula in the schema is a *subconcept*. While there exists a subconcept in the schema that has not been specialized, do the following:
  - Specialize the subconcept
  - Uniquely rename variables in the specialized description to avoid name conflicts,
  - Substitute the specialized description for the subconcept in the schema and simplify.

  Return the schema (now a fully specialized description of the concept).

Figure 4-2: The EBS Algorithm

for the target concept in the template. During subsequent problem-solving episodes, PRODIGY will maintain a utility estimate for the new rule, where utility is defined by the cumulative time cost of matching the rule versus the cumulative savings in search time provided by the rule. Only rules that prove useful are retained in the set of active control rules.

Let us return to our example problem (Figure 3-3) to illustrate how the EBS algorithm works. As stated previously, the selection of UNSTACK at Node1 provides a training example for the target concept OPERATOR-SUCCEEDS.

Target concept: (OPERATOR-SUCCEEDS *op goal node)* ,
Example*: (OPERATOR-SUCCEED8 (UNSTACK B) (HOLDING B) Nodal)

Because OPERATOR-SUCCEEDS is a recursive definition, the success of UNSTACK depends on its success at nodes 9 and 10 and eventual application at Node1 1. We shall illustrate the specialization process by "unrolling" the recursion from the bottom up, starting with the success of UNSTACK at Node11.

Subconcept: (OPERATOR-SOCCXXDI *op goal node),*
Example:(OPERATOR-SUCCEEDS (UNSTACK B)(HOLDING B) Node11)

Because UNSTACK. directly solves (HOLDING B) at Node11, OPERATOR-SUCCEEDS is specialized by Schema-Si in Appendix 1.1 - the schema consistent with the example. The system then recursively specializes the subconcepts as shown below:

Specialize (OPERATOR-SUCCEEDS *op goal node)* using Scheme-81:

(AND (ADDED-BY-OPERATOR *goal op)*
(APPLICABLE *op node)*

Specialize (ADDED-BY-OPERATOR *goal op)* using Scheme-DI:

(AND (AND (MATCHES *op* (UNSTACK *x,y)*
(MATCHES *goal* (HOLDING *x))*
(APPLICABLE *op node)*

Spaoialisa (APPLICABLE *op node)* using Schema-D4:

(AND (AND (MATCHES *op* (UNSTACK *x y)*
(MATCHES *goal* (HOLDING *x))*
(AND (MATCHES *Of* (UNSTACK *u V))*
(KNOWN *node* (AND (CLEAR *ii)* (ON *u v)* (ARMEMPTY)))

After some trivial simplifications, our result can be re-expressed as follows:

(OPERATOR-SUCCEEDS *op goal node)* if
(AND (MATCHES *op* (UNSTACK *x,y))*
(MATCHES *goal* (HOLDING *x))*
(KNOWN *node* (and (CLEAR *x)* (ON *x y)* (ARMEMPTY)))

This simply states that an operator succeeds in solving a goal at a node if the operator is UNSTACK, the goal is to be holding an object, and the preconditions of UNSTACK are known to be true at the node. Though this result is not particularly useful by itself, it serves as a lemma in explaining why the application PUTDOWN at node10 enabled the application of UNSTACK:

Sub-Concept: (OPERATOR-SUCCEEDS *op goal node) ,*
Example:(OPERATOR-SUCCEEDS (UNSTACK B)(HOLDING B) Node10)

This time, OPERATOR-SUCCEEDS is specialized by schema-S2, the recursive definition of OPERATOR-SUCCEEDS, because PUTDOWN was applied as a precursor to UNSTACK, rather than directly solving the problem. Then APPLICABLE is specialized as before, and the recursive specialization of OPERATOR-SUCCEEDS is accomplished using the result from Node11. Finally, the concept (CHILD-NODE-AFTER-APPLYING-OP *child-node pre-op node)* is specialized and then simplified; doing so effectively regresses the constraints on *child-node* across the definition of PUTDOWN. (The relevant proof-schemas used for regression are shown in Appendix 1.3). The result states that UNSTACK will succeed whenever the goal is to hold an object, and the preconditions of the sequence PUTDOWN *w,* UNSTACK *x y* are satisfied:

(OPERATOR-SUCCEEDS *op goal node)* if
(AND (MATCHES *op* (UNSTACK *x y))*
(MATCHES *goal* (HOLDING *x))*
(KNOWN *node* (AND (CLEAR *x)* (ON *xy)* (HOLDING *w)))*

The specialization process continues in this manner, until finally the successful selection of UNSTACK at Node1 is explained. The resulting expression indicates that UNSTACK is appropriate if the goal is to hold an object and the preconditions of the sequence UNSTACK *w x,* PUTDOWN *w,* UNSTACK *x y* are satisfied.[6] That is, UNSTACK is appropriate when the goal is to hold a block, there is a single block on top of the desired block, and the robot's arm is empty:

(OPERATOR-SUCCEED8 *op goal node)* if
(AND (MATCHES *op* (UNSTACK *x y))*
(MATCHES *goal* (HOLDING *x))*
(KNOWN *node* (AND (ON *x y)* (ON *w x)*
(CLEAR *w)* (ARMEMPTY)))))

At this point, a control rule can be built from the rule construction template for OPERATOR-SUCCEEDS. However, the reader may have noticed that the learned description seems very specific. Although we will not discuss here the process by which PRODIGY evaluates the utility of the rules it learns, it should be clear that the learned rule will be a relatively weak control rule. In the next section, we will see that a much better explanation of why UNSTACK was appropriate can be found by by analyzing why the alternative choice, PUTDOWN, failed[7].

This example was chosen for several reasons. First, it is similar to the USEFUL-OP example in LEX2, as described in 122], and thus can be directly compared to previous work. Furthermore, the example illustrates that learning from successful operator applications is not guaranteed to produce strong control rules. As we will see, the utility of explanation-based learning in a given task domain depends greatly on the choice of target concept and explanation.

[6] Notice that the arguments of the first UNSTACK must be *w* and *x* in order to achieve the preconditions of the remainder of the sequence. Otherwise unique variables would take the place of *w* and x.

[7] A third possibility, not discussed in this paper, is to formulate an inductive proof explaining thai UNSTACK succeeds no matter how many blocks are on top [16, 5,26].

5. Learning from Failure

To learn from the failure of PICKUP at Node1 in our example (Figure 3-3), PRODIGY uses the target concept OPERATOR-FAILS. Proof-schemas for specializing OPERATOR-FAILS are shown in Appendix 14. These schemas state that an operator fails to achieve a goal at a node if:

- the operator is rejected by a control rule, or
- the operator is not relevant to the goal, or
- the operator is not applicable and subgoaling fails, or
- the operator is applicable, and all operator applications result in failure.

The reader is cautioned that for expository clarity these schemas are a simplified subset of those used in the actual implementation.

To explain why the selection of PICKUP at Node1 failed, PRODIGY must explain why the subtree rooted at Node2 failed. As in the previous section, the analysis is recursive and corresponds to a pre-order traversal of the tree. We will describe the results of the EBS process in a bottom-up manner. First, attempting to PUTDOWN B at Node3 failed because the goal (HOLDING B) was already on the goal-stack, resulting in a goal-stack cycle. Therefore, specialization of OPERATOR-FAILS at Node3 proceeds as follows:

Target concept: (OPERATOR-FAILS op goal node)
Example: (OPERATOR-FAILS (PUTDOWN B) (ONTABLE B) Node3)

specialize (OPERATOR-FAILS op goal node) by Schema-r3:

(AND (ADDED-BY-OPERATOR *goal op)*
(IS-PRECONDITION *p op)*
(KNOWN *node* (NOT *p))*
(SUBGOALING-FAILS *p node))*

Specialize (ADDED-BY-OPERATOR *goal op)* by Schama-D2,
Specialize (IS-PRECONDITION *p op)* by 8ohana-D3,
Specialize (SUBGOALING-rAILS *p node))* by Schema-r5:

(AND (AND (MATCHES *op* (PUTDOWN *x))*
(MATCHES *goal* (ONTABLE *x)))*
(AND (MATCHES *op* (PUTDOWN y))
(MATCHES *p* (HOLDING y)))
(KNOWN *node* (NOT *p))*
(ON-GOAL-STACK *node p)))*

PRODIGY simplifies this result to the following expression, which states that PUTDOWN fails if (ONTABLE *x)* is the goal and (HOLDING *x)* is not true, but is on the goal-stack.

(OPERATOR-FAILS *op goal node)* if
(AND (MATCHES *op* (PUTDOWN *x))*
(MATCHES *goal* (ONTABLE *x))*
(KNOWN *node* (NOT (HOLDING *x)))*
(ON-GOAL-8TACK *node* (HOLDING *x))*

This expression describes why the operator PUTDOWN failed at Node3. Because PUTDOWN was the only operator relevant to (ONTABLE B) the goal at Node3, the node failed. Therefore the concept (NODE-FAILS *node)* specializes to the following, given (NODE-FAILS Node3) as the training instance, and using the result shown above as a lemma:

(MODE-FAILS *node)*
(and (IS-GOAL *node* (ONTABLE *x))*
(KNOWN *node* (NOT (HOLDXMG *x)))*
(ON-GOAL-STACK *node* (HOLDXMG x))

Moving up the tree, we find that the description above also explains why Node2 failed. In fact had the user asked PRODIGY to interleave learning with problem-solving, the inevitable failure of Node2 would have been recognized without the necessity of exploring the rest of the subtree below Node2. In effect, because PRODIGY reasons about its failures in the process of learning, *dependency-directed backtracking [7]* results when learning and problem solving are interleaved.

At this point, PRODIGY can explain why the selection of PICKUP failed at Node1. Because the precondition (ON-TABLE B) was not true, it became a goal at Node2 and (HOLDING B) was pushed down on the goal-stack, resulting in the failure described above. Thus, after the appropriate specializations, EBS yields the following learned-description for OPERATOR-FAILS:

(OPERATOR-FAILS *op goal node)* if
(AND (MATCHES *op* (PICKUP *x))*
(MATCHES *goal* (HOLDING *x)*
(KNOWN *node* (NOT (ONTABLE *x)))*

This description can then be converted into a control rule asserting that if the current goal is (HOLDING *x)* and *x* is not on the table, then PICKUP should be rejected. However, in addition, an even higher-level rule can be learned by specializing the target concept OP-IS-SOLE-ALTERNATIVE (whose definition is shown in Appendix 14), which says that an operator should be selected if all other operators will fail. The description shown above, stating why PICK-UP failed, is used as a lemma while specializing OP-IS-SOLE-ALTERNATIVE. Thus, PRODIGY learns the first control rule shown in Figure 3-4, stating that (UNSTACK *x y)* should be selected if the goal is to be holding *x,* and *x* is not on the table. (Selecting UNSTACK is slightly more efficient than rejecting PICKUP, because, as described in section 3, PRODIGY first selects the appropriate alternatives, and then uses rejection rules to filter this set. If a single alternative is selected, it is not necessary to check the rejection rules.) Notice that this rule provides a much more general and more efficient description of why UNSTACK was appropriate than the rule learned in the previous section.

As this example illustrated, learning why an alternative choice failed can be more useful than learning why a candidate succeeded [13]. This is especially true in the blocks world, where one can succinctly state the reason why a choice was "stupid". Learning from success and from failure are complementary optimization techniques whose relative utility varies from domain to domain.

6. Learning From Goal Interactions

An example of a goal interaction [30] occurs when PRODIGY attempts to solve (AND (ON A B)(ON B C)) by first stacking A on B before stacking B on C (see Figure 6-1). Of course, stacking A on B deletes (CLEAR B), a prerequisite for stacking B on C. Consequently, there is no way to achieve (ON B C) without unstacking A from B.

In the general case, we say that a plan exhibits a goal interaction if there is a goal in the plan that has been negated by a previous step in the plan. There are two complementary forms in which a goal interaction may manifest itself. A *protection violation* occurs when an action undoes a previously achieved goal, requiring the goal to be re-achieved. A *prerequisite xnolation* occurs when an action negates a goal that arises later in the planning process. In our example, a prerequisite violation occurs when stacking A on B deletes (CLEAR B). (Furthermore, a protection violation will follow when picking up B deletes the goal (ON A B). The two phenomenon can also occur independently.)

Goal interactions typically result in sub-optimal plans. Some previous planning systems have included built-in mechanisms for avoiding goal interactions [28,3,32]; PRODIGY improves on this by reasoning about interactions in order to learn control rules (in a manner reminiscent of Sussman's HACKER [30].) Since goal interactions can be unavoidable in some problems (and may even occasionally result in better plans) PRODIGY learns rules that express preferences, thereby enabling solutions to be found even when interactions are unavoidable. Learning to avoid goal interactions is an optimization technique specifically designed for planning domains; In other types of domains (e.g., theorem-proving) this technique may be irrelevant.

To explain why a decision caused a goal-interaction, PRODIGY must show that all paths in the search tree subsequent to that decision resulted in a protection violation, a prerequisite violation or a failure. Shown below is an informal definition of the target concept GOAL-CAUSES-INTERACTION which describes how selecting a goal before another goal at a node[8] can result in goal interaction:

(GOAL-CAUSES-INTERACTION *goal1 goal2 node)* if
forall paths in which goal1 is selected before goal2, either:
- the path terminates in a failure, or
- while achieving goal2, goal1 is negated (a protection violation), or
- while achieving goal2 a subgoal arises that was negated in the process of achieving goal1 (a prerequisite violation).

Figure 6-1 illustrates the relevant section of the search tree from our example. Node1 and Node6 represent the two goal orderings that PRODIGY attempts. As can be seen, it is not necessary to expand the entire tree to prove that a goal interaction occurs if (ON A B) is attacked before (ON B C). Next to nodes 2 through 5 are shown the intermediate descriptions produced by EBS in explaining why the goal interaction occurred. The target concept and training example are:

Target concept:
(GOAL-CAUSES-INTERACTION *goal1 goal2 node)*
Example:
(GOAL-CAUSES-INTERACTION (ON A B) (ON B C) Node1)

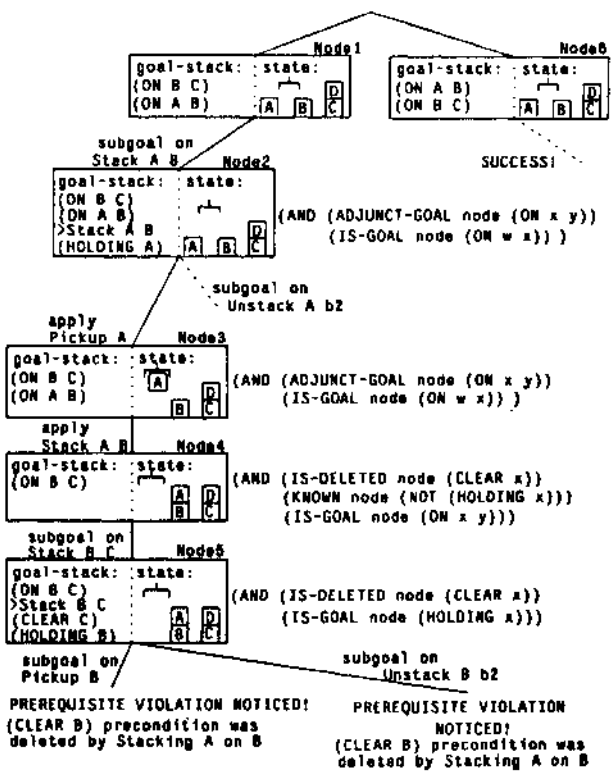

Figure 6-1: Search Tree Illustrating Goal Interaction

•The proof schemas for goal-interactions are similar to the proof schemas for failures. A goal interaction can be regarded as a "soft" failure. As with failure, the wrong choice of a node, goal, operator, or bindings can result in a goal interaction. The final result of the learning process is the goal preference rule shown in Figure 3-4 indicating that (ON *x y)* should be attacked before (ON *w x)* when both are candidate goals. After learning this rule, PRODIGY will always work from the bottom up whenever the goal is to produce an ON chain.

The key point in the learning process occurs at Node4, where EBS reveals that attacking (ON *v1 v2)* before (ON *v3 v4)* results in an interaction only if *v2* equals *v3*. This equivalence is a necessary part of the proof because achieving (ON *v1 v2)* must delete (CLEAR *v3)* for the prerequisite violation to occur. The fact that *v2* and *v3* were incidently bound to the same constant in the example is *not* the reason for the equivalence.

7. Discussion

7.1. Explanation-Based Specialization

The most obvious difference between PRODIGY'S EBS method and previous explanation-based learning algorithms, such as CBG[17], EBG [22], EGGS [23] and the STRIPS MACROPS algorithm [9], is that EBS is a specialization-based method rather than a generalization-based method. Although this is primarily an algorithmic distinction, it demonstrates how the appropriateness of an EBL algorithm depends on the environment in which learning takes place. As an example, consider the EBG method described by Mitchell, Keller, and Kedar-Cabelli. The first step of the EBG method is to create a fully instantiated explanation (in the form of a proof tree). In the second step, EBG takes the explanation and computes a general set of sufficient conditions under which the explanation structure holds. Similarly, the CBG, EGGS and STRIPS methods are given an instantiated explanation (often a sequence of operators produced by a problem solver) and they compute sufficient conditions under which the explanation structure holds. Thus these generalization-based methods all take a ground-level explanation and compute a generalization that is based on the structure of the explanation. There are two basic approaches to generating the explanation. Either it is constructed by a black box as in EBG (perhaps using a theorem prover as demonstrated by Mostow and Bhatnagar [24]), or more typically, an observed solution sequence serves as the explanation, as in STRIPS. In the latter case the problem solving operators *are* the domain theory[9]

EBS, on the other hand, never bothers to create a ground-level explanation. Instead, a generalized proof tree is expanded from the top down[10]. There are two factors that make this method appropriate. First, there is no ground-level explanation that is immediately observable, as in STRIPS. Although the problem solving trace is available, it does not constitute a useful explanation; to explain failures, goal-interactions, etc., an explanation must be constructed from the appropriate architecture-level and domain-level axioms. Secondly, mapping between from the problem solving trace to the explanation can be efficiently accomplished in a top down manner. The discrimination functions that augment the theory control the explanation process; by specifying the appropriate schema (i.e., inference rules) with which to specialize each concept

[9]These two schemes have been implemented in a variety of ways. For example, Dejong and Mooney [6] discuss a hybrid scheme in which the observed operator sequence is optional. If there is no observed operator sequence, it is constructed from the domain theory. In either case, a fully instantiated explanation is eventually produced. In the SOAR system [27], an observed sequence of production firings is the basis for learning. SOAR learns whenever a sequence of productions produces a result for a goal. (Because learning is implemented on the production level and a result is obtained independent of the goals failure or success, SOAR is able to learn from certain forms of failure as well as success.)

[10]Also, note that in EBS, unlike EBG, the learned description represents the *weakest* sufficient conditions under which the proof holds [15].

they provide all the information necessary to directly find the preconditions of the generalized proof. Therefore PRODIGY simply specializes the target concept top-down to produce the learned description, instead of first constructing the instantiated proof and then regressing the general conditions through the proof structure. In fact, the learned description produced by EBS is more general than the training example only because EBS terminates before producing the ground-level explanation. The *bias* in EBS (i.e., the factor determining the generality of the resulting description [21,31,101) comes from the proof schemas, and in particular, is determined by the disjunctive and primitive concepts used.

7.2. The Utility Of Multiple Optimization Strategies

In this paper, we have described how Explanation-based learning can serve as a general method for implementing optimization strategies to improve problem solving performance. In particular, we have illustrated four strategies for dynamically improving the performance of the PRODIGY system; each strategy corresponds to a target concept that can be explained, and therefore, be learned. In the future, more target concepts can be added to PRODIGY by augmenting the theory as necessary.

The four optimization strategies illustrated in this paper are completely general, as they are applicable in any domain specified to PRODIGY. A better question concerns the utility of the techniques. Obviously, PRODIGY can always learn from observing solutions, as do other EBL systems. However, using additional target concepts gives the system a range of options that can result in better performance, as illustrated by our examples. In general, the utility of a particular target concept depends greatly on the task domain and its formalization. In a sense, any optimization technique can be regarded as a strategy for recovering from suboptimal formalizations - those that are epistemologically adequate but procedurally inadequate [14]. The need for multiple strategies is suggested by both theoretical and practical work on program optimization [2,1].

8. Acknowledgements

We gratefully acknowledge the assistance of Craig Knoblock, Dan Kuokka and Henrik Nordin in designing and implementing the PRODIGY system. Ideas and suggestions by Jerry Dejong, Oren Etzioni, Yolanda Gil, Smadar Kedar-Cabelli, Rich Keller, Sridhar Mahadevan, Tom Mitchell, Ray Mooney, Jack Mostow, and Prasad Tadepalli greatly influenced our work.

I. Proof Schemas

The following proof schemas represent a very simplified subset of the actual schemas used in the PRODIGY implementation. We have assumed, for instance, that the preconditions of operators are simple existentially quantified conjunctions, that there can be not negated goals, and that an operator can have at most one formula in its add-list relevant to a goal. A more complete description of the actual schemas used in PRODIGY can be found in [15].

1.1. Architecture-Level Schemas for OPERATOR-SUCCEEDS

Schema-Si: An operator succeeds if it directly solves the problem.
(OPERATOR-SUCCEEDS *op god node)* if
(AMD (ADDED-BY-OPERATOR *goal op)*
(APPLICABLE *op node))*

Schema-S2: For an operator to succeed, another operator may first be required.
(OPERATOR-SOCCEEDS *op goal node)* if
(AMD (APPLICABLE *pre-op node)))*
(OPERATOR-SUCCEEDS *op goal child-node)*
(CHILD-MODE-AFTER-APPLYING-OP *chad-node pre-op node)))*

Schema-S3: For an operator to succeed, subgoaling may be necessary. (Sub goaling creates the node where pre-op applies. See Schema-S2.)
(OPERATOR-SUCCEEDS *op goal node)* if
(AMD (CHILD-NODE-AFTER-SUBGOALING *child-node pre-op node)))*
(OPERATOR-SUCCEEDS *op goal child-node)*

1.2. Some Domain-level Schemas for the Blocks World

Schema-DI: A postcondition of (UNSTACK *bl b2)* is (HOLDING B1)
(ADDED-BY-OPERATOR *goal op)* if
(AMD (MATCHES *op* (UNSTACK *bl bl))*
(HATCHES *goal* (HOLDING *bl)))*

Schema-D2: A postcondition of (PUTDOWN *b)* is (ONTABLE *b)*
(ADDED-BY-OPERATOR *goal op)* if
(AMD (MATCHES *op* (PUTDOWN *b))*
(MATCHES *goal* (ONTABLE *b)))*

Schema-D3: A precondition of (PUTDOWN *b)* is (HOLDING *b)*
(IS-PRECONDITION *subgoal op)* if
(AMD (MATCHES *op* (PUTDOWN *b))*
(MATCHES *subgoal* (HOLDING *b)))*

Schema-D4: UNSTACK is applicable if its preconditions are established.
(APPLICABLE *op node)* if
(AMD (MATCHES *op* (UNSTACK *bl b2))*
' (KNOWN *node* (AMD (CLEAR *bl)* (ON *bl b2)* (ARMEMPTY))))

1.3. Architecture-level Axioms for Computing Regressions

Schema-RI: Computes the constraints on *node* when *child-node* results from applying *op*. Note that *child-node* and *op* must be bound when this schema is used, otherwise the result may be incorrect.
(CHILD-NODE-AFTER-APPLYINO-OP *chad-node op node)* if
(FORALL *x* SUCH-THAT (KNOWN *x child-node)*
(REGRES8-ACROSS-OP-APPLICATION *node child-node x op)))*

Schema-R2: The following two schemas represent frame axioms. All literals that hold at the child node were either added by the operator, or hold at the parent node (and don't match any member *of* the operator's delete list).
(REGRESS-ACROSS-OP-APPLICATION *parent-node child-node lit op)* if
(AMD (KNOWN *child-node lit)*
(IN-ADD-LIST *add op)*
(MATCHES *add lit))*

Schema-R3:
(REGRESS-ACROSS-OP-APPLICATI0N *parent-node child-node lit op)* if
(AMD (KNOWN *x parent-node))*
(FORALL *del* 8UCH-THAT (IN-DELETE-LI8T *del op)*
(MOT (MATCHES *dd lit)))*

Schema-R4: These 3 schemas regress facts when subgoaling. Called by Schema-F6.
(REGRESS-ACROSS-SUBGOAL-ACTION *node subgoal formula)* if
(KNOWN *node formula)*

Schema-R5:
(REGRES8-ACROSS-SUBOOAL-ACTION *node subgoal formula)* if
(AMD (MATCHES *formula* (0N-GOAL-STACK *child-node god)*
(I8-GOAL *node god)))*

Schema-R6:
(REGRE8S-ACROSS-SUBGOAL-ACTION *node subgod formula)* if
(MATCHES *formula* (IS-GOAL *child-node subgoal))*

I.4. Architecture-level Axioms relevant to OPERATOR-FAILS

Schema-F1: An operator fails if it is rejected by a control rule.

```
(OPERATOR-FAILS op goal node) if
  (REJECTED-BY-CONTROL-RULE op goal node))
```

Schema-F2: An operator fails if it is not relevant to the current goal.

```
(OPERATOR-FAILS op goal node)) if
  (FORALL add SUCH-THAT (IN-ADD-LIST add op)
      (NOT (MATCHES add goal)))
```

Schema-F3: An operator fails if it's not applicable, and subgoaling fails.

```
(OPERATOR-FAILS op goal node)) if
  (AND (ADDED-BY-OPERATOR goal op)
       (IS-PRECONDITION subgoal op)
       (KNOWN node (NOT subgoal))
       (SUBGOALING-FAILS subgoal node))
```

Schema-F4: An operator fails if all possible applications result in failure.

```
(OPERATOR-FAILS op goal node) if
  (AND (ADDED-BY-OPERATOR goal op)
       (APPLICABLE op)
       (NODE-FAILS child-node)
       (CHILD-NODE-AFTER-APPLYING-OP child-node op node))
```

Schema-F5: Subgoaling fails if a goalstack cycle occurs.

```
(SUBGOALING-FAILS subgoal node) if
  (ON-GOAL-STACK node subgoal))
```

Schema-F6: Subgoaling fails if all child nodes fail after subgoaling.

```
(SUBGOALING-FAILS subgoal node)) if
  (AND (NODE-FAILS child-node)
       (FORALL formula SUCH-THAT (KNOWN formula child-node)
       (REGRESS-ACROSS-SUBGOAL-ACTION node subgoal formula))
```

Schema-F7: A node fails if solving one of the goals at that node is unsolvable.

```
(NODE-FAILS child-node) if
  (AND (IS-GOAL child-node goal)
       (FORALL op SUCH-THAT (IS-OPERATOR op)
            (OPERATOR-FAILS op goal child-node)))
```

Schema-F8: A node fails if there is a state cycle.

```
(NODE-FAILS child-node) if
  (PREVIOUSLY-VISITED-STATE child-node))
```

Schema-F9: An operator is the only possible choice if all other operators fail.

```
(OP-IS-SOLE-ALTERNATIVE unique-op goal node) if
  (FORALL op SUCH-THAT (IS-OPERATOR op)
      (OR (EQUAL op unique-op)
          (OPERATOR-FAILS op goal node)))
```

References

1. Aho, A.V., Sethi, R. and Ullman, J.D.. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

2. Blum, M. "On Effective Procedures for Speeding Up Algorithms". *Journal of the ACM 18,2* (1971).

3. Carbonell, J. G.. *Subjective Understanding: Computer Models of Belief Systems.* Ann Arbor, MI: UMI research press, 1981.

4. Carbonell, J.G. and Gil, Y. Learning by experimentation. Proceedings of the Fourth International Workshop on Machine Learning, Irvine, CA, 1987.

5. Cheng, P. W. and Carbonell, J. G. Inducing Iterative Rules from Experience: The FERMI Experiement. Proceedings of AAAI-86, 1986.

6. DeJong,G.F., Mooney, R. "Explanation-Based Learning: An Alternative View". *Machine Learning 1,2* (1986).

7. Doyle, J. "A Truth Maintenance System". *Artificial Intelligence* 12, 3 0979).

8. Ellman, T. Generalizing Logic Circuit Designs by Analyzing Proofs of Correctness. IJCAI-9 Proceedings, 1985.

9. Pikes, R., Hart, P. and Nilsson,N. "Learning and Executing Generalized Robot Plans". *Artificial Intelligence* 3,4 (1972).

10. Flann, *NS.* and Dietterich, T.G. Selecting Appropriate Representations for Learning from Examples. AAAI Proceedings, 1986.

11. Hammond, K.J. Learning to Anticipate and Avoid Planning Problems through the Explanation of Failures. AAAI Proceedings, 1986.

12. Keller, R.M. *The Role of Explicit Knowledge in Learning Concepts to Improve Performance.* Ph.D. Th., Dept. of Computer Science, Rutgers University, 1986.

13. Kibler, D. and Morris, P. Don't be Stupid. IJCAI-7 Proceedings, 1981.

14. McCarthy, J. Epistemological Problems of Artificial Intelligence. In *Readings in Knowledge Representation,* Brachman, RJ and Levesque, H.J., Eds. Morgan Kaufmann, Inc., 1985.

15. Minton, S. *Acquiring Search Control Knowledge: An Explanation-based Approach.* Ph.D. Th., Carnegie-Mellon University, 1987. Forthcoming.

16. Minton S., Carbonell, J.G., Knoblock, C.A., Kuokka, D., and Nordin, H. Improving the Effectiveness of Explanation-Based Learning. Proceedings of the Workshop on Knowledge Compilation, Oregon State University, September, 1986.

17. Minton, S. Constraint-Based Generalization. AAAI Proceedings, 1984.

18. Minton, S. Selectively Generalizing Plans for Problem Solving. IJCAI-9 Proceedings, 1985.

19. Minton, S., Carbonell, J.G., Etzioni, O, Knoblock, C.A., Kuokka, D.R. Acquiring Effective Search Control Rules: Explanation-Based Learning in the PRODIGY System. Proceedings of the Fourth International Workshop on Machine Learning, Irvine, CA, 1987.

20. Mitchell, T., Mahadevan, S. and Steinberg, L. LEAP: A Learning Apprentice for VLSI Design. IJCAI-9 Proceedings, 1985.

21. Mitchell, T. The Need for Biases in Learning Generalizations. Dept. of Computer Science, Rutgers Univ., 1980. Tech report CBM-TR-117.

22. Mitchell, T., Keller, R., and Kedar-Cabelli, S. "Explanation-Based Generalization: A Unifying View". *Machine Learning* 1,1 (1986).

23. Mooney, RJ and Bennet, S. W. A Domain Independent Explanation-Based Generalizer. AAAI Proceedings, 1986.

24. Mostow, J. and Bhatnagar, N. Failsafe -- A floor planner that uses EBG to learn from its failures. IJCAI-10 Proceedings, 1987.

25. Nilsson, N.J.. *Principles of Artificial Intelligence.* Tioga Publishing Co., 1980.

26. Prieditis, A.E. Discovery of Algorithms from Weak Methods. Proceedings of the International Meeting on Advances in Learning, Les Arcs, Switzerland, 1986.

27. Rosenbloom, P.S. and Laird, J.E. Mapping Explanation-Based Generalization Onto Soar. AAAI Proceedings, 1986.

28. Sacerdoti, E. D.. *A Structure for Plans and Behavior.* Elsevier Publishing Co., 1977.

29. Silver, B. Precondition Analysis: Learning Control Information. In *Machine Learning, An Artificial Intelligence Approach, Volume II,* Morgan Kaufmann, 1986.

30. Sussman, G. J.. *A Computer Model of Skill Acquisition.* Elsevier Publishing Co., 1975.

31. Utgoff,P.E. *Shift of Bias for Inductive Concept Learning.* Ph.D.Th., Rutgers University, May 1984.

32. Vere,S.A. Splicing Plans to Achieve Misordered Goals. IJCAI-9 Proceedings, 1985.