In Minton, S. (Ed.). *Machine Learning Methods for Planning*. San Mateo, CA: Morgan Kaufmann, 1993.

CHAPTER 15

A Method for Biasing the Learning of Nonterminal Reduction Rules

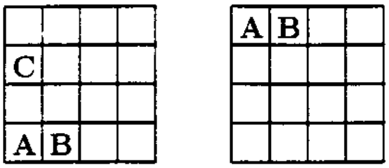
STACY C. MARSELLA
CHARLES F. SCHMIDT

1. Introduction

The focus of our research is on experiential learning within a problem reduction architecture. The first question that we address is that of motivating and developing an appropriate criterion against which to evaluate what is learned. Experiential learning systems are often evaluated using a model of efficiency that is defined directly on the basic operations of the specific performance system that utilizes the learning. We will follow a different course. We will develop an ideal model of efficiency for problem reduction and use the model as the criterion against which to evaluate what is learned. This approach has two advantages. First, because the model is abstract, it can be applied to the evaluation of learning within any performance system that utilizes problem reduction. Second, this abstract model of efficiency serves as the basis for our design decisions in implementing our learning and performance system.

The second question that we address is that of the specific design and preliminary test of a learning component (PRL) designed to satisfy this criterion. The crucial feature of this design is the ability to control the formation of hypotheses concerning appropriate problem decompositions. We will then contrast PRL's performance with the performance of an analogous learning component, BU-PRL. Unlike PRL, this learning component can control the formation of hypotheses in only a limited way.

We begin with a very simple problem of the sort that our learning system will be presented. Figure 15-1 provides such an example problem. As the figure shows, the domain consists of a square grid of locations, some of which are occupied by tiles denoted by a letter—A, B, or C in this case. The primitive moves that are allowed in this domain consist



Starting State Goal State

Figure 15–1. An example sparse sliding-tile problem.

Table 15–1. The Representation of the Example Problem in Figure 15–1

```
                 PROBLEM:
STARTING STATE:  location of tile A is cell k
                 location of tile B is cell l
                 location of tile C is cell m
                 upadjacent to cell k is cell o
                 cell o is clear
                 rightadjacent to cell k is cell l
                 ...
    GOAL STATE:  location of tile A is cell p
                 location of tile B is cell q
```

of moving a single tile up, down, left, or right to an immediately adjacent location. A move can be made if there is an immediately adjacent location in the direction of the move and if that location is not already occupied by some other tile.

Our learner begins with knowledge of these primitive moves, the immediate adjacency relations between locations, and the locations of the tiles. The learner is not given information concerning any global properties that might hold in the problem domain. For example, it does not know that the locations can be organized into rows and columns, that if any tile is in a corner location and has two tiles that are adjacent, then it cannot be moved without first moving another tile, and so on. Table 15–1 describes the example problem using the initial domain language presumed throughout this chapter.

The example problem is solved when the problem solver discovers some sequence of primitive moves that leave A and B in the upper row as depicted in Figure 15–1. Note that the remaining tile, C, is not constrained to a particular location in the goal state of the problem.

This problem is not difficult to solve. There are literally thousands of ways of solving it, and our problem solver could discover any one of these by using its knowledge of the primitive moves as a basis for a systematic search. The hard problem is not to find a solution, but to find a solution that will provide a basis for determining what to learn from this example problem.

One answer to the problem of solution choice is to learn from a solution that solves the problem with a minimum number of moves. In our example problem this ability narrows the choice from thousands of possibilities to five possible solutions. But it turns out that this minimal moves criterion is the wrong criterion to apply to learning within the context of a problem reduction problem solver. To see why, we will review the basic characteristics of problem reduction search and consider a way in which to accommodate planning within this search strategy.

2. Problem Reduction Search

Problem reduction search realizes a solution by recursively decomposing a problem into subproblems until "primitive" subproblems are recognized (Amarel, 1984: Nilsson, 1971). A problem or subproblem consists of a pair $< S0, G0 >$ where the first element of the pair is the starting state for that subproblem and the second element is the goal state to be achieved. The search to be carried out in problem reduction can be represented as an AND/OR-tree with the AND-branches representing decompositions of the parent subproblem and the OR-branches representing alternative decompositions. A search for a solution is successful if a subtree exists in this AND/OR-tree whose root is the root of the tree. whose branches are all AND-branches, and whose leaf nodes are all primitive subproblems. We will refer to this subtree as the *effective derivation* of a solution. The representation of the entire search for a solution to a particular problem is referred to as the *derivation.* A derivation is said to be deterministic if it is equivalent to the effective derivation. This is the case when no OR-branches exist in the derivation and a solution has been obtained. Clearly, a derivation that is deterministic is more efficient than one that involves some degree of nondeterminism.

A deterministic search is guaranteed if the information required to solve each subproblem can be completely specified when the parent subproblem is expanded and if all subproblems at each level of the tree can be expanded in parallel or, equivalently, in any order. When a pair of subproblems at a level in the tree can be expanded in parallel, then the pair is typically said to be independent. Note that the pair may not be independent in the logical sense, but only functionally independent in the sense that both can be completely and correctly specified at the point at which they are expanded. This strictly top-down deterministic search represents one ideal model of efficiency for problem reduction search. However, planning problems never exhibit such complete independence. Consequently, we next consider the way in which dependencies can be efficiently managed in problem reduction search.

Problem reduction search involves the application of two types of rules. One type, the nonterminal rules, serve to decompose subproblems into an ANDed set of subproblems. The other type, the terminal rules, are the recognizers of primitive subproblems. In our application of problem reduction to planning problems, the terminal rules provide the basis for representing the primitive moves or actions of the domain. For example, the movement of tile A from its location in the starting state to the location immediately above is a primitive move that would be recognized as a primitive subproblem.

A plan is typically defined as a valid ordering of the primitive moves. In our application of problem reduction, the actions of the plan correspond to the terminal nodes of the effective derivation. However, the effective derivation does not place an ordering on these terminal nodes. Consequently, the method must be extended in some way to yield a plan, that is, a valid ordering on the actions represented by the terminal nodes.

Dependency between nonprimitive subproblems is typical of planning problems. Consequently, in order to extend this method of search to planning, we must also consider the way in which such dependencies can be recognized and accounted for in the problem reduction search. In our example problem, a nonterminal rule might decompose the problem into two subproblems, one whose goal is to achieve tile A in its goal location and the other whose goal is to achieve tile B in its goal location. The starting state for each subproblem might be the initial state of the problem. In this case, there exists no solution to the problem where these two subproblems are independent.

Subproblem dependency can be appropriately resolved if the order in which the subproblems should be solved is known. The appropriate ordering allows the information or constraints introduced by the solution of one subproblem to be passed to the context that is needed to solve the dependent subproblem. At each level of the AND-tree of an effective derivation there is an implicit partial order on the expansion of the subproblems that allows the search to be carried out deterministically. We will refer to this correct partial order on expansion at each level of the tree as the *planning order.* If the planning order for each decomposition in the effective derivation is known and can be used to control the expansion of subproblems, then a deterministic derivation can be guaranteed. However, a mechanism must be added that passes constraints up the tree of subproblems whenever an ordering of subproblem expansion is required.

Analogously, dependencies between actions can be resolved by imposing an appropriate ordering on the actions that are dependent. An ordering on the execution of actions will be referred to as an *execution ordering.* The planning and execution ordering are logically distinct. For example, consider again our sample problem. Figure 15–2 illustrates three distinct solutions to the example problem. The topmost is labelled the Go-Around solution. The labelled arrows shown on the grid represent the primitive actions involved in a particular solution. To the right of this grid, the labels of the primitive actions are repeated, and here the directed edges between nodes represent direct action enablement. For instance, in the Go-Around solution, action a1 directly enables a2, and a3 directly enables b1. This enablement graph also constitutes a partial order on the actions (transitive edges are not depicted). For example, a path from a1 to b1 in the Go-Around solution means that a1 must precede b1. Analogous representations are depicted for what are termed the Toggle solution and the Clear-Out solution. These latter two solutions both involve primitive actions that move tile C from its initial location, whereas the Go-Around solution involves no movement of tile C. Note that for both solutions that involve the movement of C, it is necessary to plan the solution to the A and B subproblems prior to planning the solution for the movement of C. The reason is that the location to which C is moved depends on how the A and B subproblems are solved. However, the actions that realize the movement of C must come before some of the actions that solve the A and/or B subproblems. Thus, both planning and execution ordering are required to deterministically specify a derivation of a solution that involves subproblem dependencies.

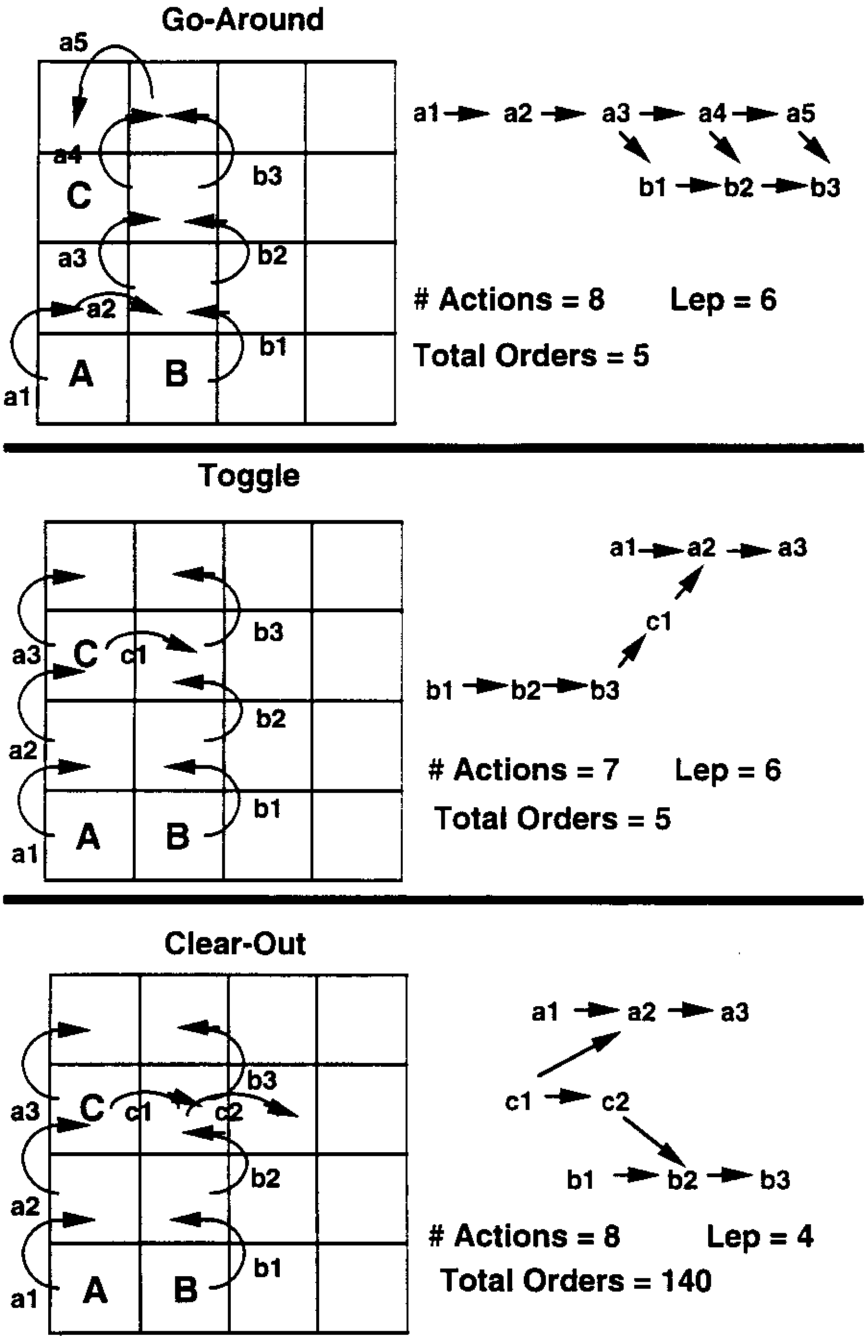


Figure 15–2. Different solutions to the example problem of Figure 15–1.

The ordering on actions may appear to be required only at the terminal nodes of the effective derivation. However, as we will develop in more detail, the partial order on the actions that are depicted for the Go-Around as well as those for the Clear-Out solution cannot be represented in tree form. A (nondegenerate) tree induces a partitioning of the actions, and there is no partitioning of these actions that can be ordered in a way that captures the generality of the partial order associated with these solutions. This implies that if the most general partial order that is consistent with a set of actions is to be represented, then an additional process and representational machinery must be added to the basic problem reduction mechanism. This added process is needed to examine and annotate the set of terminal nodes associated with an effective derivation of a solution.

The necessity for this additional process can be avoided if we restrict the representation of the permissible partial order of a solution to a partial order that can be represented by specifying an execution ordering at each level of the AND-tree. We call a partial order that can be represented in this fashion a *hierarchical partial order* (HPO). Figure 15–3 shows an HPO for the actions of the Clear-Out solution. Clearly, we have sacrificed some generality in the partial order. Now, all of the actions moving tile C must precede those that move tiles A and B. But we can now specify the execution ordering at the level where the problem is decomposed into subproblems involving tiles A, B, and C. When both the planning order and execution order can be correctly specified at each level of decomposition, then we can guarantee that (1) the derivation is deterministic, and (2) the ordering of the actions can be recovered directly from the effective derivation. Thus, in this case the only process that has been added to the special case in which subproblems and actions are both completely independent is a process that passes constraints between dependent subproblems, that is, subproblems whose order of expansion is specified.

If the correct planning and execution orderings for a solution can be represented at each level of the tree that constitutes the effective derivation of a solution, then we take this abstract characterization of a deterministic derivation to be the most efficient derivation that can be realized using the problem reduction method when the solution involves subproblem and/or action dependencies. We can refer to this model of efficiency as the HPO model. If a learner can be biased to acquire solutions whose planning and execution orderings satisfy this HPO model,

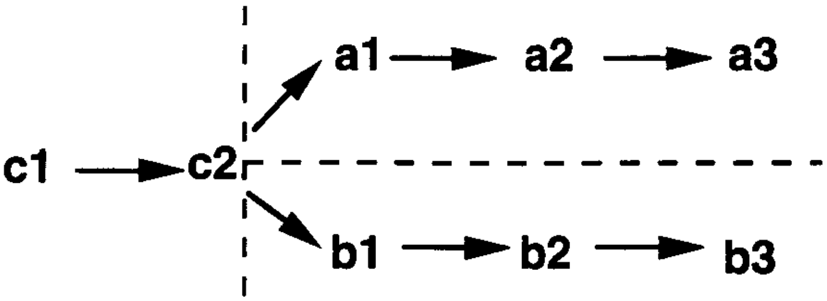


Figure 15–3. An HPO on the actions of the Clear-Out solution.

then any measure of the learner's performance in solving problems covered by the learning must be more efficient than that of a learner that does not satisfy this HPO model.

The HPO model serves to partition the space of possible derivations for a problem into those whose effective derivation is representable as hierarchical partial orderings on the levels of the tree and those that are not so representable. Within the space of those effective derivations that satisfy the HPO model, one such derivation may differ from another in the degree to which an ordering must be imposed, or, equivalently, in the degree of dependency that holds between the various subproblems or actions that occur in the derivation. Solutions that exhibit a simpler structure of dependencies are preferred. The necessity to order subproblem expansion incurs the cost involved in passing information up the derivation tree. Similarly, if constraints are generated via the simulation of terminal actions, then the ordering of actions that must be simulated also increases cost. We refer to the measure that we have developed to reflect the simplicity of the planning and execution ordering as LEP. The measure involves determining for the terminal actions in a solution the longest dependency path that holds in the associated partial order, that is, the cardinality of the maximum chain in the partial order. In general, LEP can range from one to the number of terminal actions. If there is no ordering relation in the partial order, then the value of LEP is one. Conversely, if there is only one valid ordering of the solution's k terminal actions, then the LEP is k.

Let us return to Figure 15–2 to illustrate the calculation of this measure. Recall that the partial orders illustrated in this figure for the three solutions reflect the existence of a direct enablement relation between the pair of actions connected by a directed arc. The Clear-Out solution has a LEP of four, whereas each of the other two solutions has a LEP of six. Also computed for each of these solutions is the number of total orders that are consistent with the solution's partial order structure. There are 140 such total orders that are consistent with the partial order represented by the Clear-Out enablement graph. Only five total orders are consistent with the enablement graph for the other two solutions. Thus, what might appear to be small differences in LEP can reflect very large differences in the number of total orders.

The LEP measure provides us with an additional measure against which to evaluate the solutions that are learned. We can also obtain a measure of the longest dependency path involved in the planning of a solution. In this case, the partial order on the terminal elements is obtained by projecting onto these elements the planning orders that are annotated in the derivation tree.

3. A Problem Reduction Planning System

REAPPR (Bresina, Marsella, & Schmidt, 1987) is a problem reduction system that allows the user to specify partial orders on derivation and execution when expressing a nonterminal reduction rule and to use such information effectively. In REAPPR a problem is specified as a pair of descriptions $<S0.G0>$, where the first specifies the problem's starting state and the second the goal state. The descriptions are allowed to be partial state descriptions in the sense that there may be a set of state pairs in the underlying state space that satisfy these descriptions. In order to effectively use the planning and execution orders specified in the nonterminal rule, REAPPR carries out what we term *full problem decomposition* when it applies a nonterminal rule. This term was chosen to contrast with goal decomposition. In full problem decomposition the nonterminal rule provides the information needed to decompose the problem into a set of subproblem pairs, $<S1, G1>, <S2. G2>, \ldots, <Sk, Gk>$. Thus, the parent goal is decomposed into a set of goals, and the parent's starting state may undergo a decomposition.

This expressive power allows a nonterminal rule to essentially predictively specify characteristics of "islands" in the solution path. This capability has been explored in various planning and design tasks, including the tower of Hanoi (Bresina, Marsella, & Schmidt, 1987), the sparse sliding-tile domain (Marsella, 1988), and a music composition task (Marsella & Schmidt, 1988).

Search is realized in PRL's planner via the application of nonterminal and terminal rules. The main syntactic features of a nonterminal rule are depicted in Table 15–2. The left-hand side of the rule is a description of the problems to which the rule can be applied. The right-hand side of a nonterminal rule includes four components: the decomposition structure, the planning order, the execution order, and a local evaluation function. The decomposition structure is a list of labelled subproblems, each of which consists of a reference label and a partial specification of a starting state and a goal state. The rule's planning order determines the order in which subproblems are expanded. If some subproblem, A, is ordered before another, B, then A must be fully expanded (i.e., a complete subtree) until all its terminal subproblems are achieved before B can be expanded. The execution order determines the order in which solutions to subproblems must be combined to achieve a partial order on the achievement of actions. Thus, if the solution to some subproblem A must precede the solution to subproblem B, then every action in A must be ordered before every action in B. Both planning and execution orders are of the form $ordering - scheme = (order\ term_1\ \ldots\ term_k)$, where order is either *seq* (a linear order, or chain, of $term_1\ \ldots\ term_k$) or *par* (the $term_1\ \ldots\ term_k$ are unordered, i.e., an antichain) and where each $term_i$ is either one of the subproblem labels in the decomposition structure (and thus a child node in the tree) or any embedded ordering scheme.[1] Every subproblem label must occur exactly once in each order.[2] The expression (seq ... $term_i$ ... $term_k$ ...) means that any subproblem whose label is part of $term_i$ is ordered before any subproblem whose label is part of $term_k$, whereas (par ... $term_i$ ... $term_k$...) means that any subproblem whose label is part of $term_i$ is not ordered with respect to any subproblem whose label is part of $term_k$. Finally, there is a local evaluation function. The learning system places restrictions on the form of this function to facilitate the manipulations that occur during learning, but this issue will not be discussed here.

[1] A par execution order is to be interpreted as any total order sequence of the $term_i$.

[2] Strictly speaking, this requirement does not apply to the execution order but is constrained to be the case.

Table 15-2. REAPPR's Nonterminal Rule Form

Left-Hand Side of Rule

- **Applicability Test:**
  - **Problem:**
    - **S0:**
      - **partial state spec.**
    - **G0:**
      - **partial state spec.**

Right-Hand Side of Rule

- **Decomposition Structure**
  - **List of subproblems each of the form:**
    - **Label**
    - **Starting State: partial state spec.**
    - **Goal State: partial state spec.**
- **Planning Order:**
  - **Partial Order over subproblem labels**
- **Execution Order:**
  - **Partial Order over subproblem labels**
- **Evaluation Fcn:**
  - **lambda form**

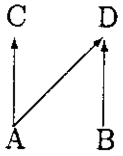
The terminal rules have a left-hand side whose syntax is identical to the nonterminal rules. The right-hand side contains the representation of the action as well as a local evaluation function.

Extended problem reduction search proceeds in three phases: *node selection*, *node expansion*, and *tree re-marking*. The selection phase picks a node in the AND/OR-tree to expand based on the planning order annotations in the tree and on the values returned by the local evaluation functions. The planning orders determine a set of nodes that can be expanded next, and the evaluation functions determine the "best" node from this set to expand. The expansion phase determines which rules apply to the node. For those rules that apply, the match context of the rule's applicability test is used to resolve the right-hand side of the rule, which then is used to form a new node connected by an OR-branch to the node being expanded. The search can be parameterized to limit the number of OR-branches formed, in terms of both the number of rules that are applied and the number of different ways in which any particular rule is applied. A cost is associated with each new OR-branch node by applying the rule's local evaluation function to the node being expanded. If, on the other hand. no rule applies to the node that was selected for expansion, then that node is marked with an infinite cost. Either way, the search re-marks the cost value in the tree and returns to the selection phase. The cost of an OR-branch is the minimum cost of the children, and the cost of an AND-branch is the maximum cost of its children. Marking with an infinite cost will force the system to find an alternative node to expand during selection. The effect of setting an infinite cost is to follow an alternative AND-tree in the AND/OR-tree (i.e., backtrack). The choice of alternative AND-tree is guided by the local cost functions, planning strategies, and overall structure of the tree.

When the search forms a complete AND-tree, it terminates successfully. If, however, the search reaches a point where the root node has infinite cost, then every alternative AND-tree has a node with infinite cost, and the values have percolated to the top of the tree. An impasse results.

4. On the Representation of Partial Orders in Plans

The annotations of partial orders within an AND-tree's hierarchical structure realize a partial order on the terminal nodes of the tree. For example, consider an AND-tree, consisting of a set of nodes N, that is annotated with partial orders on planning and execution. For simplicity, let us first consider just one of the annotations, say the execution order—the same structural characteristics hold for the planning order, although the order need not be the same.. Let N_i denote some node in the tree and T_i denote the set of terminal nodes of the subtree rooted at N_i. Given the root node, N_0, in the tree, let P_0 be the partial order on execution *realized* on the terminal nodes, T_0, of the annotated subtree rooted at N_0. Let $N_1, \ldots, N_k$ denote the children of N_0, and let E_0 be the partial order on execution *annotated* in N_0 and thus defined over the set $N_1, \ldots, N_k$. Thus, E_0 is a partial order on N_0's children, specified



$a \prec c \,\&\, a \prec d \,\&\, b \prec d \,\&\, a\,\mathcal{I}\,b \,\&\, c\,\mathcal{I}\,d \,\&\, b\,\mathcal{I}\,c$

Figure 15–4. The Z partial order.

by an *ordering-scheme.* If E_0 orders N_j before N_k, then P_0 orders every element of T_j before every element of T_k.[3] In this way, the annotations in the AND-tree effectively express a *hierarchical partial order*, a partial order that can be realized by the annotations in the AND-tree.

We noted above that not every partial order is hierarchical. For instance, the partial order whose Hasse diagram is depicted in Figure 15–4 is not hierarchical. Because of its shape, we call this the Z partial order. This is the "simplest" partial order (fewest edges and nodes) that is not hierarchical. The Z partial order has special relevance as the basis for recognizing whether some arbitrary partial order is an HPO. A partial order, P, on a set, A, satisfies the *Z-less* condition if the partial order does not have any subgraphs isomorphic to the Z partial order. More precisely, a partial order is Z-less if the following is false for any subset of A and restriction of P to that subset: $a\,\mathcal{I}\,b \,\&\, c\,\mathcal{I}\,d \,\&\, a \prec c \,\&\, a \prec d \,\&\, b \prec d \,\&\, b\,\mathcal{I}\,c$, where $\mathcal{I}$ is the associated incomparability relation of the partial order. Although it will not be demonstrated here, the absence of a "Z" subgraph can be used to recognize whether a partial order is hierarchical.

Although a hierarchical partial order cannot represent every partial order,[4] it can obviously represent the extremes of any total order and any antichain. For those orders that cannot be represented, there is always an extension to the partial order that is hierarchical, as suggested by the various ways the Z-less condition can be satisfied—$a \not\mathcal{I} b$, $c \not\mathcal{I} d$, or $b \not\mathcal{I} c$.

Using the Z-less condition, we can reexamine the enablement graphs of Figure 15–2. The Toggle solution's enablement graph satisfies the

[3] In more traditional terminology, P_0 can be expressed by the *composition* graph $P_0 = E_0[P_1, \ldots, P_k]$, where $P_1 \ldots P_k$ are the partial orders realized by the subtrees rooted at $N_1 \ldots N_k$, respectively, the children of N_0. In such a graph, E_0 is termed an external factor and the $P_1 \ldots P_k$ are internal factors.

[4] Indeed, its partial order dimension cannot be greater than 2.

Z-less condition and is an HPO. However, both the Go-Around and Clear-Out graphs are not HPOs. In the case of the Go-Around, the subgraph consisting of nodes a4, a5, b1, and b2 cause the graph to fail the Z-less condition. Extending the partial order with the inclusion of an arc from b1 to a5 transforms it into an HPO that is consistent with four of the five total orders of the original graph. In the Clear-Out, there are two subgraphs isomorphic to the Z partial order. One consists of the nodes a1, a2, c1, and c2. The other consists of nodes c1, a2, b1, and b2. Figure 15–3 depicts an extension to the original partial order that, by ordering c2 before a1 and b1, realizes an HPO that is consistent with 20 of the original 140 total orders.

Recall that the HPO model also requires that the planning partial order annotation be specified within the tree's hierarchy. As a consequence a node's planning order annotation is specified over the same nodes as the execution order (and as a consequence over the same equivalence classes of nodes denoted by the children of that node.). Otherwise, the node expansion and solution composition phases of the reduction search could not be represented within the same tree. The effect is that the structure of the tree shared by the two order annotations constrains the HPOs that can be represented in that tree. However, in the trivial case in which the tree has a maximum depth of 1 (every node is either the root or a terminal) there is no constraint—any HPO on planning and any HPO on execution can be expressed by the respective annotations.

5. Two Approaches to Learning

Learning is addressed as an integrated component of the problem reduction planner. The planner receives as input an ordered sequence of problems that it attempts to solve in order. Prior and future problems in the sequence are not known, and the system has no model of the class of problems defined by that sequence. In trying to solve a problem, the planner can (1) fail to find a complete AND-tree, (2) find a tree that because of dependency does not have a valid solution, or (3) find a complete tree that is a valid solution. The first two cases result in opportunities to learn. The failure to generate a complete AND-tree for some problem in the sequence results in an impasse, and control is passed to the learning component, which receives as input from the planner some nonterminal node on the frontier of the incomplete AND-tree, along with the set of existing rules. The learning task is, then, to acquire a new nonterminal reduction rule to expand that node. The failure to generate a valid solution results in the modification of a rule that is used in the derivation in order to avoid the dependency. In either case, learning is based on only the existing rules and a node in the tree that failed.

The main criterion placed on the learning and modification of these rules is the degree of dependency in both the derivation of the plans and the execution of the solutions in those plans. This focus on degree of dependency suggests the proposition that learning nonterminal rules can be viewed in terms of restrictions on the actions taken by solutions to the subproblems: that is, independence between subproblems can be achieved by restricting the actions taken to solve the subproblems. Given that the restrictions can be characterized in some way that suggests subproblems, a search in the space of restrictions can be used to realize a biased search of a space of alternative decomposition hypotheses and, therefore, nonterminal rules.

Two problem reduction learners, PRL and BU-PRL, have been designed and implemented to evaluate this restriction-based approach to biasing the learning of nonterminal rules. Both systems use a refinement-based approach to learning whereby an initial rule can be modified by feedback from problem-solving experience. The distinction between PRL and BU-PRL is that PRL uses a hypothesis-driven approach to forming that initial rule. In particular, PRL searches a space of alternative decomposition hypotheses to derive a tentative rule hypothesis. That decomposition space is searched indirectly by searching a closely coupled space, a space of alternative restrictions on solution paths in the state space. By determining the initial rule, this search in a space of alternative path restrictions realizes feed-forward control over the structure of the rule and, therefore, control over the refinements that can result through the use of that rule.

In BU-PRL, this hypothesis-driven search for the initial rule is replaced by a search in the space of solutions for a minimal length solution to the problem. From this solution, an initial decomposition rule is formed. This search is in itself not meant to be interesting; in essence, it emulates receiving a problem solution from an expert. The significant distinction between BU-PRL and PRL is that BU-PRL explores the opposing proposition that a desirable plan structure need not be coerced, but rather is derivable from a state space solution.

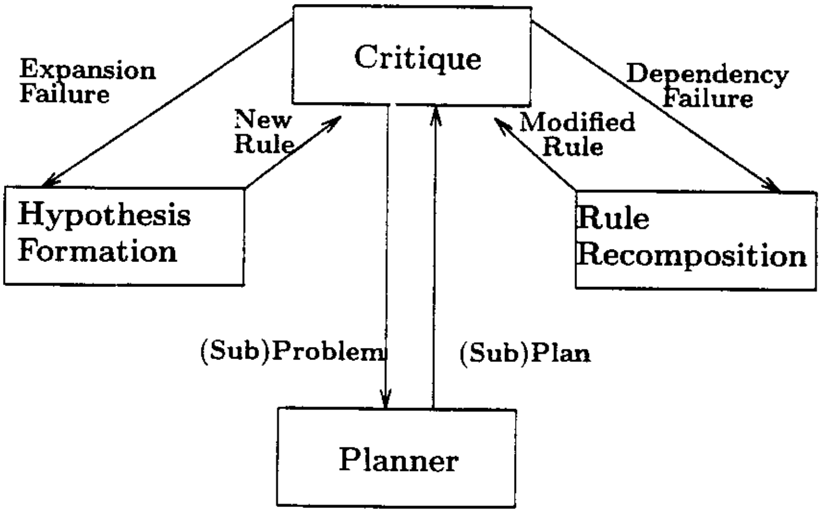


Figure 15–5. The major components of the PRL architecture.

5.1 An Overview of PRL

Figure 15–5 depicts the major components of PRL. The central component is Critique, which, at the top level, sequentially receives problems to be solved. When invoked, Critique tests whether a problem can be solved by passing it to the problem reduction planner. The planner attempts to generate a complete AND-tree and returns control to Critique. Depending on the state of this tree, Critique then passes control to Hypothesis Formation or Rule Modification, or it requests a new problem to work on.

If the tree is not complete, Critique selects a subproblem in the tree that could not be expanded and passes control to Hypothesis Formation, which derives its name from the fact that it uses a hypothesis-driven approach to forming rules. When control is passed to Hypothesis Formation, it attempts to generate an ordered list of nonterminal decomposition rule hypotheses for this subproblem if such a list does not already exist. If a nonempty list is generated (or already exists), Hypothesis Formation returns the first item of the list to Critique and stores the remaining hypotheses; otherwise, it returns failure. If a hypothesis is returned, Critique recursively invokes itself to test whether this subproblem can be solved using the rule set embellished by the new rule hypothesis. Otherwise, it is the case that all the hypotheses have been exhausted and Critique exits. If this is a top-level invocation of

Critique, this exit results in failure and a request for the next problem in the sequence. If it is not a top-level invocation, then Critique exits to a higher level invocation of Critique and reinvokes Hypothesis Formation.

On the other hand, if Critique gets a complete AND-tree from the problem reduction planner, the dependencies in the AND-tree are analyzed to see whether the plan is correct. If the plan is correct, Critique returns success and a potentially embellished rule set. It then requests a new problem. If the plan is not correct, Critique selects an incorrectly solved subproblem and the rule that expanded that subproblem. It then classifies the error in that subproblem's solution as involving some form of dependency, and Critique passes control to Rule Modification, which returns either failure or a modified version of the rule. If failure is returned, Critique exits; otherwise, Critique is recursively invoked on the subproblem with the rule set embellished by the modified rule.

To summarize, each of the various components of PRL has a specific role. When Hypothesis Formation is presented with a problem that cannot be solved, it uses the problem to realize a systematic bias on the consideration of decomposition hypotheses prior to any complete search for a solution using one of these hypotheses. That complete search, performed by Critique and the planner, tests the hypothesis, which leads to its acceptance, its rejection, or its modification by Rule Modification.

5.2 An Example of Learning in PRL

With the above as an overview, we turn to a discussion of a nonterminal rule being hypothesized and modified for the example problem of Figure 15–1. For this example, we assume that there are no nonterminal rules in the initial rule set and that the only terminal rules are those that represent the primitive moves (i.e., the permissible state space operators). Thus, presenting the example problem to the system will immediately result in an invocation of Hypothesis Formation on that problem. The discussion will concentrate on Hypothesis Formation since it is how PRL biases learning and thus distinguishes PRL from BU-PRL.

5.2.1 HYPOTHESIS FORMATION

Hypothesis Formation proposes restrictions on the paths in the state space taken by solutions and, based on those restrictions, it proposes a nonterminal decomposition rule for the impasse subproblem. The

Table 15–3. Major Steps of Hypothesis Formation

- Perform limited-depth backward partial search.
- Perform limited-depth forward partial search.
- Apply pairing and decomposition heuristics.
- Form rule based on preferred pairing set.

major steps of Hypothesis Formation are depicted in Table 15–3. In the implementation, these steps are not undertaken in a strictly sequential fashion. In particular, the search for a hypothesis is wedded closely to heuristics designed to circumvent the potentially large hypothesis space and bias the search of that space. However, for explanatory purposes, we have teased these steps apart in order to simplify the description.

Hypothesis Formation starts with a backward search from the goal, followed by a corresponding forward search from the starting state. Both are made to a limited depth (in this case a depth of one in both directions) using existing rules.[5] In the present example, this "bidirectional search" would involve applying the terminal rules that represent the up, down, right, and left operators backward from the goal state and forward from the starting state.

Figure 15–6 depicts this bidirectional search on the example problem. N_f and N_b denote the nodes on the forward and backward search frontiers, respectively. Each node is marked by the action taken and the resulting change in tile location. The expression "Loc A 41" signifies that the location of tile A is cell 41, where cell 41 denotes the fourth row and first column, in effect, the top-left cell. This row and column format is used throughout this example simply to ease readability; recall that the system has no such global knowledge of the grid structure.

The backward search uses partial matches between the goal (or adds) expressed in the rule to the desired goal state (Nilsson, 1971). Partial matches are also allowed in the forward direction under the following conditions. The expressions that must match are those expressions with

[5] Either terminal rules or nonterminal rules can be used, but recall that in the present example we are restricting ourselves to terminal rules that correspond to primitive operators.

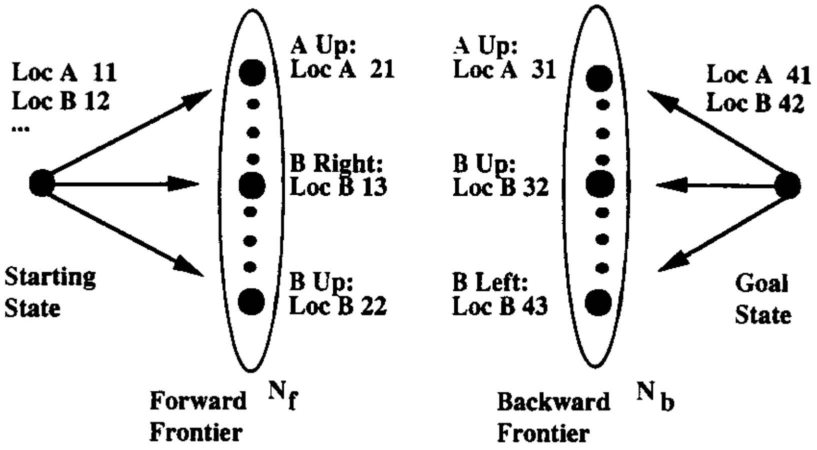


Figure 15–6. Partial bidirectional search with nodes on the frontiers marked by the action taken and the resulting tile location.

fixed relations, that is, relations that cannot be deleted or added by some operator (here, any adjacency relation), as well as those *dynamic* (i.e., not fixed) expressions that are in the goal state and that persist in the corresponding backward search frontier, which in this example are the location relations. Other dynamic relations need not match (here, the clear relations). This constrained partial match in the forward direction serves to replicate the goal-directed behavior of the backward direction so as to restrict the solution structures that the searches suggest.

Now, consider elements of the Cartesian product, $N_f \times N_b$, of all possible pairings of the nodes, one from each frontier. Each pairing constitutes an incomplete search, or disconnected paths in the state space.[6] These pairings are the bases for suggesting restrictions on complete paths in the state space that may solve part of the goal. Furthermore, subsets of $N_f \times N_b$ constitute a basis for an alternative decomposition of the goal. An important prelude to forming a decomposition is to form restrictions on the individual pairings. These restrictions play critical roles in (1) biasing the search through the space of alternative decompositions, (2) forming a generalized decomposition hypothesis, and (3) determining the conditions under which a decomposition is tested.

[6] Under the assumption that the search used terminal rules.

The approach to deriving restrictions on a complete path in the state space from a pairing's incomplete search path involves consideration of the expressions tested in reaching a pairing's nodes on the search frontiers. These "match contexts" are used to tie nodes from the backward search to nodes from the forward search. For instance, consider tying together the forward node Up A and the backward node Up A in Figure 15–6. Note that the expressions denoting the location of tile A are tested in both directions and that they persist in the resulting states in both directions. This persistence in both directions leads to the restriction that the same tile always be moved. Further, the rules involved are both up moves, which suggests a restriction that each action employed to generate the full path always be a Move Up.

Based on the above restrictions, *pairing heuristics* then order the pairings, preferring the most restrictive pairings. For instance, pairings in which each search direction uses the same rule are preferred. Also preferred are pairings with similar forward and backward match contexts. For our example, pairings that move the same tile in the forward and backward search directions would be preferred. Note that the purpose of these heuristics is to enforce a preference on those pairings that suggest the greatest restriction on the solution (and match context) for a subproblem.

Decomposition heuristics are also applied to prune subsets of pairings from consideration and establish a preference on the overall decomposition structures. For instance, subsets of $N_f \times N_b$ are required to cover the goal, and the preference is for those subsets (1) that actually partition the goal, (2) in which the match contexts of the N_f nodes in the subset do not overlap, and (3) that have lower cardinality. Note that (2), and to a lesser extent (3), are quite the opposite of the pairing heuristic that preferred restrictions, or overlap, in a pairing. In terms of the sliding-tile domain, covers whose match contexts did not overlap regions of the board would be preferred.

When these two sets of heuristics are applied to the impasse problem, the most preferred subset of pairings would consist of two pairings, an up move applied to tile A in both directions and an up move applied to tile B in both search directions. A tentative hypothesis about the decomposition structure is now formed, based on the rules used to reach the frontiers in each pairing of the preferred set, the match contexts of these two pairings, and the pairing restrictions. The other, less preferred pairing sets are stored, since the preferred pairing subset may fail

Table 15–4. Representation of Initial Rule Hypothesis Formed by Hypothesis Formation

Problem:		**Subproblem1:**
S0: location ?tile1 is ?x1		**S1: location ?tile1 is ?x1 . . .**
location ?tile2 is ?x2	→	**G1: location ?tile1 is ?z1**
. . .		**Subproblem2:**
G0: location ?tile1 is ?z1		**S2: location ?tile2 is ?x2 . . .**
location ?tile2 is ?z2		**G2: location ?tile2 is ?z2**

Planning Order: unordered
Execution Order: unordered

Evaluation Fcn: good rule if following aggregates are true
aggregate upadj ?x1... ...?z1 aggregate upadj ?x2... ...?z2

Key: ?name = match variables
. . . = 0 or more values or expressions

the subsequent Critique phase. Table 15–4 depicts the rule that PRL hypothesizes for the present example. Its decomposition structure has two subproblems consisting of a straight upward path for one tile and a straight upward path for the other. The *problem* is the rule's applicability test, or left-hand side. To the right of the applicability test are the subproblems of the decomposition, and below these elements are the planning order and the execution order. Finally, at the bottom of the figure is the evaluation function, which we discuss below.

The subproblems and the left-hand side of the rule are formed from the left-hand sides of the rules used to generate the two pairings in the preferred set. For example, variables in starting and goal states of the Move Up rules are renamed consistent with the restrictions. Thus, the same variable denotes the tile in the starting and goal states, but no similar constraint is placed on the cell location.[7] These starting and goal

[7] An alternative approach to forming this generalized rule would be to use standard goal regression or explanation-based generalization approaches to form the preconditions for the subproblems. Since the restrictions tie together expressions from the forward and backward partial searches for each pairing, in this case via the location of tile expressions, preconditions from the backward search can potentially be regressed to the starting state. In the example, the significant difference between these approaches would be that the upadjacent relation that allows the final up movement to the goal location for each pairing would also be included in the starting state.

states then become the subproblems of the new rule's decomposition and are composed to form the left-hand side of the rule. The match contexts are used to distinguish the dynamic relations so as to further restrict the solution structure. Those dynamic expressions that were not tested in the backward search are excluded from the new rule. Finally, because there was no overlap in the match contexts for the pairings, the execution and planning orders are heuristically assumed to be unordered.

However, note that the subproblems in the rule's decomposition make no reference to straight upward paths. There is no reference, because the bidirectional search was incomplete, because paths are not part of the original state specification language, and because the rule matcher has no way of restricting access to the regions of the board (the columns of cells) that such paths will traverse. For these reasons, the rule is overgeneralized. More problematic is the lack of a guarantee that the supposition about a straight upward path suggested by the original pairing will actually be realized, and therefore tested, when this decomposition rule is used. This gap between what the pairing suggested and what the existing state specification language permits must somehow be closed.

To fill this gap, one additional rule, called a *simple generator*, is formed for each pairing. In addition to controlling the condition under which the testing of the new decomposition rule occurs, simple generators provide a procedural approach to realizing relations that are not in the original state description language. Table 15–5 provides a simplified depiction of the simple generator that would be formed from either pairing in our preferred pairing subset (i.e., up moves of tile A or up moves of tile B). As in the decomposition rule, this simple generator's applicability test, or left-hand side, specifies when the rule *can* be applied. The rest of the rule includes the recursive structure and the evaluation function. The recursive structure consists of two steps. Move Up moves the tile up one location, and Recurse is a recursive step.[8] Finally, the evaluation function specifies when it is most *useful* to apply the rule.

The formation of a simple generator is similar to the formation of the decomposition rule. It hinges on the restrictions that arise from the pairing. However, the simple generator is also defined so that it searches

[8] Note that the steps have the form of a subproblem's state pairs, since a simple generator is implemented as a nonterminal rule. The recursion of this rule terminates via a generic terminal rule in the system's rule base that checks whether subgoals are true in the initial state.

Table 15–5. Representation of Simple Generator Formed by Hypothesis Formation

```
Problem:                          Move Up:
 I0: location ?tile1 is ?x0        I1: location ?tile1 is ?x0
     upadj to ?x0 is ?x1               upadj to ?x0 is ?x1
     ...                               clear ?x1 ...
                           ──────> G1: location ?tile1 is ?x1
                                  Recurse:
 G0: location ?tile1 is ?xn        I2: location ?tile1 is ?x1 ...
                                   G2: location ?tile1 is ?xn
```

Evaluation Fcn: good rule if following aggregates are true
aggregate upadj ?x0... ...?xn

Key:
?name = match variables
. . . = 0 or more values or expressions

a space that is restricted and abstracted. In defining this abstracted space, PRL employs a simple heuristic approach that uses information from the pairing's match context. In particular, the starting state is abstracted by removing those expressions from the recursion context for which it can be ascertained that they will not be tested during the search. In the present example, such expressions include the right, left, and down adjacency relations since the generator is always moving the tile upward. More importantly, the backward search is used to focus the abstraction of dynamic relations. Dynamic relations in the preconditions that are not goal specific are asserted but not tested. Accordingly, the Clear relation is not tested in the left-hand side of the generator in Table 15–5. However, it is asserted in the Move Up step, and the upadjacent relation is tested. Furthermore, because the location of tile A was tested in the backward search, it is not ignored.

Although not depicted in Table 15–5, the simple generator is designed so that expressions in the preconditions and outcomes of the underlying rules used to form it (in this case the domain operator Up) are aggregated, as the generator recurses, into expressions we term *aggregates*. The determination of what is aggregated and how it is aggregated is made when the generator is formed and based on the restrictions (see Marsella & Schmidt, 1991 for additional discussion). For this Up simple generator, the specific clear cell and upadjacent relations that must be true to move a tile up a particular path are aggregated as the Up move is applied. Furthermore, the clear cells that are the outcome of the Up move are also aggregated.

This reformulation into aggregate expressions is used in two ways. The aggregates that are based on dynamic expressions are used during critiquing to test the simple generator. Those that are based on fixed expressions are used by the planner, which has mechanisms for storing and combining the fixed aggregate expressions that result from solving problems. These expressions are the basis for the testing done by the evaluation functions of the rules depicted in Table 15–4 and Table 15–5. The aggregations of fixed expressions state that the corresponding rule is likely to be particularly good[9] if it is *known* at the time of rule application that there are aggregate expressions of upadjacent cells—in essence, columns—that include the initial and goal cell locations. For instance, "aggregate upadj ?x1... ...?z1" expresses that there is a sequence of upadjacent relations in which the first relation has a first argument of ?x1 and the last relation has a second argument of ?z1. Recall that the representation of the domain did not assume or specify such global properties. Whether such aggregate expressions are true is not part of the original domain specification and must be derived from the solutions to previous problems/subproblems and remembered. The utility of this "incremental reformulation" depends on the system discovering, for the set of problems it solves, a parsimonious and uniform set of solution structures that operate in restricted spaces; for example, "straight up paths." However, a full discussion of this particular issue is beyond the scope of this paper.

5.2.2 CRITIQUE

With the completion of Hypothesis Formation's formation of the rule of Table 15–4, control is returned to Critique, which in turn invokes the problem solver, which uses the decomposition rule and simple generators to search for a solution. As the two Up generators run on tiles A and B, respectively, they aggregate the preconditions that had to be true as well as the outcomes. In this example, the simple generators succeed, returning a complete AND-tree to Critique, which tests for subproblem dependency.

Critique's detection and classification of subproblem dependency is based on the state pair descriptions at particular levels of the tree (not

[9] The term *particularly good* is used to distinguish this evaluation function from the applicability test, which is a binary, or yes/no, decision.

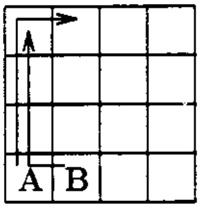


Figure 15–7. Sliding-tile problem solved by a follow-the-leader strategy.

on the lowest level action sequences). The execution order annotations throughout the tree establish whether the partial starting and goal states in the various nodes of the tree will be (or can be) coincident. If two states are coincident but not consistent, then the plan fails the critiquing test.

Thus, critiquing verifies the assumptions that are implicit in the decomposition. With this approach to critiquing, it is useful to distinguish two sources of dependency—*intersubproblem dependency* and *intrasubproblem dependency.* Intersubproblem dependency is a dependency between the state descriptions posited by the subproblems' decomposition structure. In the example, there is no intersubproblem dependency, which is evidenced by the absence of overlap between the simple generator's paths for tile A and tile B. Indeed, avoiding such a dependency was a major goal of the pairing and decomposition heuristics.

In a sliding-tile domain, intersubproblem dependency occurs when two subproblems generate paths that cross, as in Figure 15–7. In this figure, tiles A and B have goal locations in the top row of the grid, but their column positions are interchanged. Some of the ways to deal with such a dependency are to avoid it, exploit it, or push it lower in the tree. Avoidance could involve either forcing the use of different paths, which would likely require sequencing the planning of subproblems to ensure derivation of nonintersecting paths, or sequencing the execution of the subproblems' solutions. Alternatively, such dependencies can sometimes be exploited. For instance, if two paths cross, perhaps the shared path can be used for both tiles, as in the follow-the-leader solution depicted in Figure 15–7. Finally, the decomposition can be modified to push the dependency lower in the tree.

Intrasubproblem dependency is dependency between a subproblem and the parent problem's starting state or goal state. In the example, critiquing does detect an intrasubproblem dependency since the aggregated clear preconditions for tile A are violated in the actual starting state of the world—tile C is blocking tile A's path.

5.2.3 Rule Modification

Based on the critiquing, Rule Modification is invoked to modify the rule's decomposition and partial orders. The result is a modified decomposition structure: A new subproblem, call it a clearing subproblem, is spliced onto the decomposition. The modified rule is shown in Table 15–6. This new subproblem establishes the goal of making the aggregated clear relations true. Note the annotations that control the ordering of actions, the ordering of plan node expansions, and the flow of state specification information. The terms referred to as aggregate forms in subproblem1, the clearing subproblem, encapsulate a region of the state space subtended by the sibling subproblems in the decomposition that generate the motions of the goal tiles. The planning order ensures that these aggregate forms are specified when the clearing subproblem is expanded, even though the actual clearing actions must be executed prior to the movements of the goal tiles that define the region to be cleared.

This clearing subproblem illustrates the use of planning order in specifying the reduction of problems into subproblems. Clearing could be described in terms of the goal *location of blocking tiles off the paths of A and B.* Yet, this subproblem has *no* correspondent in the original goal specification. It is not expressible purely by manipulation or partition of the syntactic expressions in the original problem expressions. Furthermore, the task that is involved in solving that subproblem is ill-defined at the time the rule is used to expand the reduction search because what constitutes a "blocking tile" and where it has to be moved are relative to the paths taken for A and B. Control of planning order enables procedural derivation for the specification of the region that has to be cleared. The expression of that region depends on the formation of expressions, not in the original problem description, tailored to describe the class of solutions (i.e., the AND-subtrees) that are consistent with the overall decomposition structure and partial orders on planning and execution. Note that this is a rather sophisticated strategy, despite, and perhaps by virtue of, the simplicity of the underlying solution structure.

Two alternative modifications deserve mention. One approach is for Rule Modification to leave the rule of Table 15–4 as is and form a new rule that has a decomposition structure of two subproblems, the new

Table 15–6. Representation of Rule after Being Critiqued and Modified

Problem: **I0: location ?tile1 is ?x1** **location ?tile2 is ?x2** **...** **G0: location ?tile1 is ?z1** **location ?tile2 is ?z2**	→	**Subprob1:** **I1: location ?tile1 is ?x1** **location ?tile2 is ?x2 ...** **G1: aggregated clear expressions** **from subprob2 and subprob3** **Subprob2:** **I2: from I0** **G2: location ?tile1 is ?z1** **Subprob3:** **I3: from I0** **G3: location ?tile2 is ?z2**
Planning Order:	$(2 \prec 1)\&(3 \prec 1)$	
Execution Order:	$(1 \prec 2)\&(1 \prec 3)$	
Evaluation Fcn:	**good rule if following aggregates are true**	
aggregate upadj ...?x1...?z1...	**aggregate upadj ...?x2...?z2...**	
Key: ?name = match variables **... = 0 or more values or expressions**		

clearing subproblem and the original problem solved by Table 15–4. We present the rule as above to simplify the discussion. Another modification is for the modified rule just to clear tile A's path. A subsequent attempt to solve the problem would then detect a dependency with tile B's path, and the rule would again be modified so that it would end up as in Table 15–6. At present, Rule Modification does not take this alternative, because the resulting decomposition structure and fixed dependency scheme of clearing just tile A's path are more complex and would require breaking apart the original decomposition structure of Table 15–4. However, this approach is in some respects preferable since it does not make the stronger assumption, or requirement on the number of clear cell locations, that clearing can be achieved simultaneously for both subproblems.

In general, critiquing the AND-tree structure and the modification of rules is a rich topic that has been considered only briefly. The repairs suggested above manage dependency by restructuring and reordering the decomposition structure. The restructuring that can occur in PRL includes the grouping of dependent subproblems; the grouping of independent subproblems and the sequencing of them prior to dependent subproblems; and the splicing of an additional subproblem for failed preconditions. This restricted set of restructuring operators are designed to ensure that the planning and execution order imposed by a decomposition does not become arbitrarily complex and, of course, that it remains a hierarchical partial order. Further, the desire is to form rules that will result in placing manageable dependencies high and less easily manageable dependencies low in the tree (i.e., localized). In this domain, such restructuring also results in the collecting together of tile goals, an instance of which is presented in the next section. Thus, another form of aggregation is created, an alternative to cell location aggregation by simple generators as described above.[10]

Regardless of the particular modifications that are undertaken, the major effects on the rule hypothesis are the same. Both the original rule and its modifications must be within the constraints imposed by the HPO model. And because of this relation between Hypothesis Formation and Rule Modification, Hypothesis Formation need not directly consider all possible decompositions.

5.3 BU-PRL

Evaluation of PRL's approach to learning extended problem reduction rules is being conducted on several fronts. For instance, we are interested in evaluating whether and when a hypothesis-driven, or feed-forward, approach to learning nonterminal rules is preferable to an example-driven, or feedback, approach. To this end, we built a comparison system, BU-PRL (*B*ottom *U*p PRL), whose architecture is depicted in Figure 15–8. By design, BU-PRL shares all the major components of PRL except Hypothesis Formation, which is replaced by an Expert component.

When invoked by Critique on some problem that cannot be expanded, Expert generates a minimal length solution to the problem and then forms a tentative decomposition that has a subproblem for every terminal action in the solution. In keeping with the focus of learning rules with low dependency between subproblems, the strong assumption is made that both planning and execution strategies are unordered.[11] Thus, the rule by construction generates a complete plan but, because of the strong assumption about execution order, critiquing typically

[10] See Marsella (1988) for additional discussion of aggregation and Flann (1989) for a similar view.

[11] By virtue of these assumptions, the task of forming a general rule from the terminal rules that constitute the actions is straightforward.

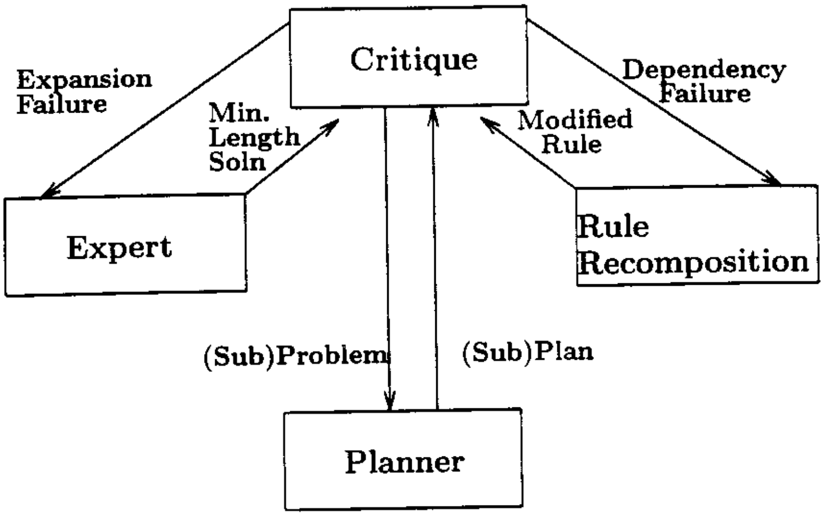


Figure 15–8. The major components of the BU-PRL architecture.

detects intersubproblem dependency. Rule Modification then restructures the decomposition, typically coalescing dependent subproblems and transforming the execution and planning strategies, under the constraint that the original solution's total order is a member of the total orders suggested by the transformed rule.[12] Control returns to Critique, which tries to solve the original problem using this modified rule. Since there may not be existing rules for the newly coalesced subproblems in the decomposition, this modified rule may not lead to a complete AND-tree when it is used. In such a case, Expert is reinvoked to form rules for the subproblems. Eventually, this process results in a rule set (with at least one new rule) that includes the original solution provided by Expert in the space of solutions it can generate.

6. Preliminary Evaluation of PRL and BU-PRL

For illustrative purposes, some preliminary data from PRL and BU-PRL are presented in Figure 15–9. At the top of the figure, BU-PRL's performance is depicted on the problem of moving tiles A and B four

[12] The original decomposition is based on terminal actions. An alternative would be to use terminal rules and any existing nonterminal rules as the basis of the original decomposition or its modification. The approach here attempts to maximize independence irrespective of existing rules.

cells straight up, with tiles C and D blocking the direct movement of tile A. BU-PRL's Expert subsystem generates 52 different solutions, 14 of which are total order variations of a Go-Around strategy and 38 of which are total order variations of a Toggle strategy. These solutions are depicted on the two tile grids at the top of Figure 15–9. Although the enablement graphs are not depicted in the figure, both solutions have enablement graphs with LEPs of seven.

The decomposition that BU-PRL realizes differs depending on which of the 52 solutions Critique and Rule Modification receive from Expert, even for solutions belonging to the same "strategy." This variation is attributable in part to the constraint that a decomposition must cover, or be able to generate, the solution provided by Expert and in part to the fact that the realization as a decomposition with fixed partial orders restricts the underlying enablement structure. Depicted underneath each tile grid is a partitioned enablement graph that represents the solution/decomposition with the weakest partial order (in the sense of having the most total order extensions) realized by BU-PRL on the corresponding solution. The dotted lines in the graphs denote the partition that the rule's decomposition achieves. Thus, for the Go-Around example, the decomposition results in three subproblems: (1) the first four movements of tile A in sequence, (2) an ensuing subproblem that realizes the last two movements of tile A in parallel with the first two movements of tile B, and (3) finally, a subproblem that realizes the last two movements of tile B. Rule Modification has grouped together dependent subproblems and independent subproblems. An example of the former is a1... a4 grouped into P1 in the Go-Around decomposition, and an example of the latter is b1b2 into P2 and a5a6 into P3 of the same decomposition. The solutions that both decompositions realize have six total order extensions and a LEP of eight. In contrast, a Clear-Out solution/decomposition that PRL learns is depicted at the bottom of Figure 15–9. It is consistent with 420 total orders and has a LEP of six.

Although these empirical investigations are at a preliminary stage, we can make some distinctions between BU-PRL and PRL. Note in Figure 15–9 that the decompositions that BU-PRL learns break up solutions into segments that have similar independence or dependence properties in the underlying state space. Being driven by example solutions, as opposed to being driven by a hypothesis, BU-PRL cannot impose global restrictions on the part of the state space that is to be searched for a solution. In addition, the resulting AND-tree that is passed

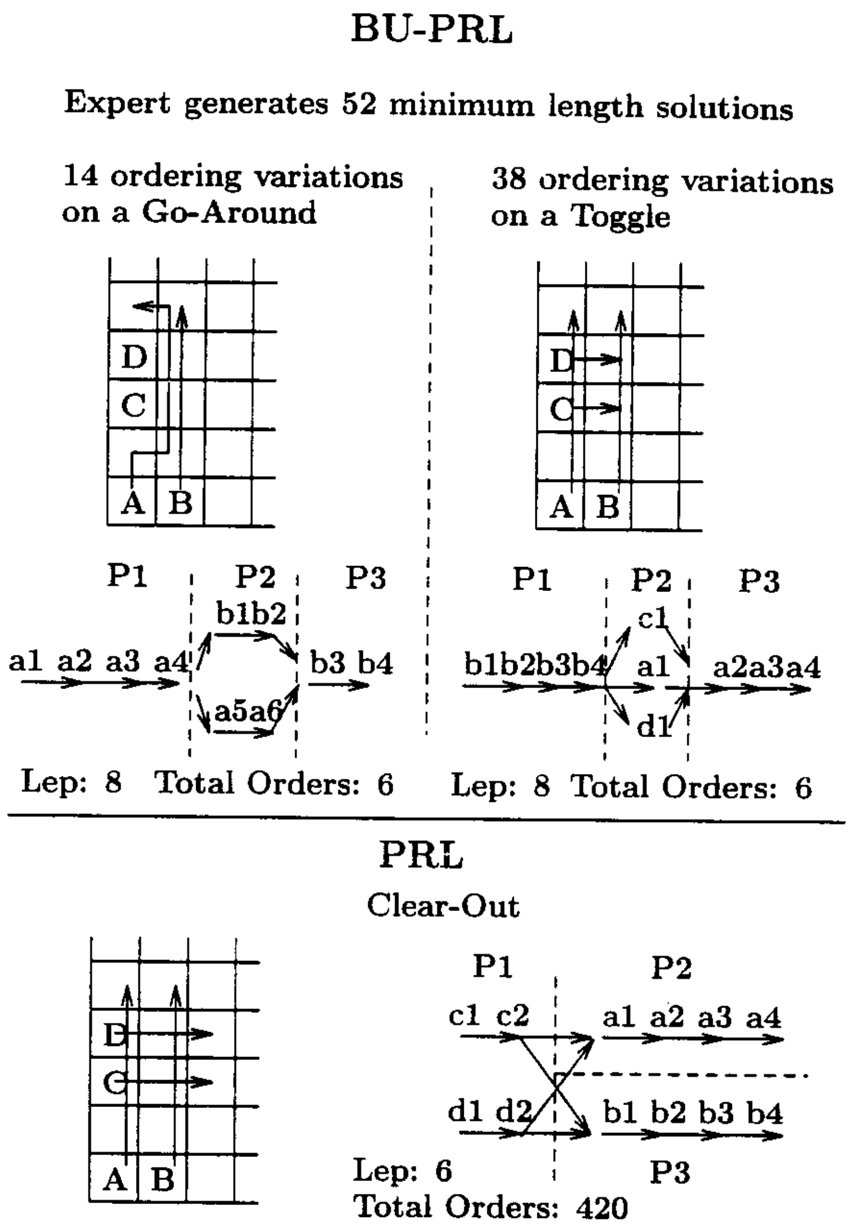


Figure 15–9. Preliminary comparisons of PRL and BU-PRL.

to Critique includes only a weak hypothesis about how the actions in that solution are partitioned; recall that each action is initially assumed to be an independent subproblem. Therefore, the significant restrictions on the modifications of a rule are that the LEP be as low as possible and that the modified rule "cover" the original solution's actions and ordering. Given that it does not exercise control over the solution from which it is learning, BU-PRL can thus derive decompositions that partition the movements of particular tiles in rather complex ways. Also as a consequence, BU-PRL tends to end up with not only a nonterminal rule for the problem but also a corpus of rules for the various local configurations of tiles that form the partitions. Such a consequence is suggested by the partitions in Figure 15–9.

In contrast, PRL can impose restrictions on both the region of the state space to consider for each subproblem and the overall decomposition structure. Thus, in the Clear-Out, there are two subproblems, one for each tile in its respective column. And because these restrictions are imposed top down, they are embedded in the structure of the AND-tree that is passed to Critique and thus constrain the modifications performed during rule modification to manipulations involving those subproblems. Therefore, even though the enablement graph of the final solution has no isolated subgraphs that would imply independence, the initial hypothesis of independence between the movements of tiles A and B guides the resulting modification toward retaining that independence.

To obtain similar behavior from BU-PRL would require a more powerful Critique and Rule Modification and a relaxation of the condition that modified rules generate the original solution. Of course, it is a considerable (but not impossible) relaxation for BU-PRL to derive a Clear-Out decomposition from a Go-Around solution. However, it is more interesting to characterize further how the two approaches differ in their present forms. To this end, we extend comparisons of PRL and BU-PRL to consider their performances on different sequences of problems and different permutations of those sequences.

For instance, we have not discussed the two systems' different approaches to learning rules with highly dependent decompositions as well as rules that do not realize a "syntactic" decomposition of the goal. An example is a "planning island" approach to moving a tile along an L-shaped path where there is one subproblem for each straight-line leg of the path and the planning island is the cell where the paths cross. For BU-PRL, such a planning island is not, by itself, problematic, or even distinguishable from other learning situations. As long as BU-PRL is provided a solution, it will learn something. The issue is whether what it learns is a "good" thing to learn. PRL is far more selective. Although we do not include the details in this chapter, PRL's Hypothesis Formation only considers such rules when rules already exist for the subproblems that will be generated and PRL can determine how to express/handle the dependency. For instance, learning such an L-shaped path rule requires that the problem solver already have acquired, from previous problem solving, rules for moving along each leg of the path to and from the planning island, which in this case is the crossing point of the two legs of the path, as well as the knowledge that the island exists. Given the limited, and very local, initial knowledge of the sliding-tile domain that the system has in the present example, this latter point means that the system must have traversed that island cell location before and knows its relation to the starting and goal locations. This rather stringent requirement on the learning of dependent decompositions follows from a desire to constrain the arbitrary chaining of rules as the basis for forming a decomposition hypothesis. In addition, because of the dependency between the subproblems, such rules are at the "bottom" of Hypothesis Formation's search.

Another issue that is not included in our discussion of PRL's Hypothesis Formation but that impacts its learning is whether the simple generators should remain in the rule set after critiquing and if so under what conditions. At present, we are testing PRL under the condition that the simple generators remain in the rule set if the Critique/Rule Modification phases are eventually successful and if the rules used in the bidirectional search that generated the corresponding pairing were the same terminal rule. In general, leaving the simple generators in the rule set can have considerable impact because Hypothesis Formation is invoked at impasse when there are no applicable rules and it hypothesizes new rules based on existing rules.

Finally we are investigating several domain-related issues. The tile domain and the representation of it in this chapter have several notable properties: The operators are invertible; an operator's deletes are a subset of its preconditions; although the domain representation is very local, the problems always present a grid with uniform adjacencies; and the space of alternative operators is rather impoverished. These characteristics impact in various ways on the nature of subproblem dependencies that arise and on Hypothesis Formation's ability to impose restrictions on the regions of the state space that are considered for its decomposition hypotheses. Exploring the system's performance in modified tile worlds. such as ones with immovable objects. holes, or irregu.ar adjacencies, as well as worlds with richer operators can further clarify the impact of these properties on problem reduction.

7. Concluding Remarks

The PRL problem-solving architecture provides an effective approach to reduction search within the HPO model of efficiency. The nonterminal rule's decomposition structure includes full problem decomposition and partial orders on planning and execution. When employed in the reduction search, these features provide a mechanism for predictively managing the dependency between subproblems within the hierarchy of the AND-tree and characterize the dependence, or degree of independence. in some "subregion" of the underlying state space associated with a problem.

However. as we noted in the various solutions to an example sliding-tile problem. alternative solutions to a problem can differ in the degree to which they can be captured in such nonterminal rules. It may not be possible to represent the various paths in the state space that are denoted by a solution's enablement graph within the partial state descriptions and hierarchical partial orders of the nonterminal rules. Furthermore. the enablement graph and the dependency it implies for any partition or decomposition can impact the utility of doing a decomposition. the generality of the resulting rule, and the weakness of the partial orders expressed in that rule.

This sensitivity of the decomposition to the specific solution poses a challenge to the learning of nonterminal rules. As we saw in our discussion of BU-PRL, learning from an example solution presumes not only that an example is of a structure and form that allow analysis but also that the example's structure provides a basis for forming a desirable strategy for the problem solver's architecture.[13] In the case of problem reduction, if the learning is to acquire the Clear-Out strategy, then an arbitrary solution, even one of minimal length, can obfuscate the kind of knowledge the system needs to acquire. The circuitous routes of the Go-Around strategy provide a good example. A dependency, or enablement, analysis will be based on a solution that has the "wrong" structure and that fails to make explicit such knowledge as the planning order.

[13] Similarly, the concern of the CRITIC in LEX (Mitchell, Utgoff, & Banerji, 1982) for minimal length was related to the problem-solving architecture.

The learning in PRL uses a hypothesis-driven heuristic method for incremental learning of nonterminal rules, as opposed to BU-PRL's solution-driven approach. The learning in PRL first forms a rule hypothesis, tests it by using it to generate a solution, and then, on the basis of the test, may modify the hypothesis. Whereas PRL's testing and potential modification of the rule hypothesis is, like BU-PRL's learning, a form of example-driven or feedback learning, PRL's Hypothesis Formation phase can be characterized as feed-forward learning. The bias introduced by Hypothesis Formation is *not* motivated by a desire to realize directly an efficient search for some solution path in the state space (as in Sacerdoti, 1974; Knoblock, 1990). Rather, it is an attempt to derive hypotheses about solution *structures* that are consistent with the problem-reduction framework and admit general methods of managing dependency within that framework. These hypotheses distinguish PRL from BU-PRL by imposing a desirable global structure on the decomposition and thus providing a guide to subsequent learning.[14]

In forming a rule hypothesis and controlling how it is tested, the learning is coerced to consider those subregions of the state space that are likely to have desirable independence properties in the reduction space. The approach presumes that when there exists a general method of handling the dependency within this reduction framework, there will be a relationship between the dependency expressed in a rule's decomposition structure and the connectivity between states in an underlying region of the state space. We expect that the learning, by biasing itself to characterize such subregions, will not only learn "good" rules for the present but also help the system realize a useful cumulative bias on sequences of problems and thus potentially provide the problem-reduction learner an alternative to estimating rule utility (Minton, Carbonell, Etzioni, Knoblock, & Kuokka, 1987).

The preliminary experiments with PRL and BU-PRL in the sparse tile domain have so far been consistent with this view. The utility of PRL's approach is tied to the characteristics of the state space graph. Clearly, in a domain in which the connectivity in state space is very sparse (in effect, highly dependent solutions) or domains that have state space graphs dominated by "narrows" (Amarel, 1968), strongly ordered solutions will be the norm and other approaches to problem solving and

[14] Gemplan's (Lansky, 1986) structuring of events into larger units presents a related view, although the motivation is not based in learning.

learning (e.g., macros) may be better. At the other extreme, in domains in which the state space is completely connected, any order will work and managing dependency is not an issue. The state space for such a domain may have uniform properties that let it be globally characterized. However, we suspect that there is a large class of domains in which the state space has a variegated structure that makes the techniques we are exploring of interest.

The PRL problem-solving architecture, with its mechanisms for managing dependency by loosely coupling subproblem searches, is complex, even for a reduction system. There has been little research on learning for such complex problem-solving systems, and the reasons are clear: Given some arbitrary solution, it is very difficult to infer an appropriate mapping to the processes and structures of the architecture. Feed-forward learning techniques may well be a necessity in such architectures.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. DCR83-18075. Our thanks to Charles Chubb for his help in defining Z-less partial orders and for the valuable discussions.

References

Amarel, S. (1968). On representations of problems of reasoning about actions. In D. Michie (Ed.), *Machine intelligence 3*. Edinburgh, Scotland: Edinburgh University Press. (Reprinted in B. L. Webber & N. J. Nilsson (Eds.), *Readings in artificial intelligence*. Palo Alto, CA: Tioga, 1981.)

Amarel, S. (1984). Expert behavior and problem representation. In A. Elithorn & R. B. Banerji (Eds.), *Artificial and human intelligence*. Amsterdam: North-Holland, 1984.

Bresina, J. L., Marsella, S. C., & Schmidt, C. F. (1987). *Predicting subproblem interactions* (Technical Report No. LCSR-TR-92). New Brunswick, NJ: Rutgers University, Laboratory for Computer Science Research.

Flann, N. S. (1989). Learning appropriate abstractions for planning in formation problems. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 235–239). San Mateo, CA: Morgan Kaufmann.

Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 923–928). Menlo Park. CA: AAAI Press.

Lansky, A. L. (1986). A representation of parallel activity based on events, structure, and causality. Reasoning about actions and plans. *Proceedings of the 1986 Workshop* (pp. 123–159). San Mateo, CA: Morgan Kaufmann.

Marsella, S. C. (1988). *An approach to problem reduction learning* (Technical Report No. LCSR-TR-110). New Brunswick, NJ: Rutgers University, Laboratory for Computer Science Research.

Marsella, S. C., & Schmidt, C. F. (1988). *A problem reduction approach to automated music compcsition* (Technical Report No. LCSR-TR-115). New Brunswick, NJ: Rutgers University, Laboratory for Computer Science Research.

Marsella, S. C., & Schmidt, C. F. (1991). *PRL: A problem reduction learner that manages dependency* (Technical Report ML-TR-32). New Brunswick, NJ: Rutgers University, Laboratory for Computer Science Research.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569). AAAI. San Mateo, CA: Morgan Kaufmann (distributor).

Minton, S., Carbonell, J. G., Etzioni, O., Knoblock. C. A., & Kuokka, D. R. (1987). Acquiring effective search control rules: Explanation-based learning in the PRODIGY system. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 122–133). San Mateo, CA: Morgan Kaufmann.

Mitchell, T. M., Utgoff, P. E., & Banerji, R. (1982). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 1). Palo Alto, CA: Tioga.

Nilsson, N. J. (1971). *Problem-solving methods in artificial intelligence.* New York: McGraw-Hill.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*, 115–135.