Integrating Planning and Learning: The PRODIGY Architecture [1]

Manuela Veloso (veloso@cs.cmu.edu)
Jaime Carbonell (carbonell@cs.cmu.edu)
Alicia Pérez (aperez@cs.cmu.edu)
Daniel Borrajo [2] (dborrajo@fi.upm.es)
Eugene Fink (eugene@cs.cmu.edu)
Jim Blythe (jblythe@cs.cmu.edu)

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213-3891
Phone number: (412) 268 2565

Running head: Planning and Learning: The PRODIGY Architecture.

In Journal of Experimental and Theoretical Artificial Intelligence, 7(1), 1995

[1]This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government. The third and fourth authors were supported by fellowships from the Ministerio de Educación y Ciencia of Spain.

[2]Current address: Facultad de Informática, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, Boadilla del Monte, 28660 Madrid (Spain).

Abstract

Planning is a complex reasoning task that is well suited for the study of improving performance and knowledge by learning, i.e. by accumulation and interpretation of planning experience. PRODIGY is an architecture that integrates planning with multiple learning mechanisms. Learning occurs at the planner's decision points and integration in PRODIGY is achieved via mutually interpretable knowledge structures. This article describes the PRODIGY planner, briefly reports on several learning modules developed earlier along the project, and presents in more detail two recently explored methods to learn to generate plans of better quality. We introduce the techniques, illustrate them with comprehensive examples, and show prelimary empirical results. The article also includes a retrospective discussion of the characteristics of the overall PRODIGY architecture and discusses their evolution within the goal of the project of building a large and robust integrated planning and learning system.

1 Introduction

The PRODIGY architecture was initially conceived by Jaime Carbonell and Steven Minton, as an Artificial Intelligence (AI) system to test and develop ideas on the role of machine learning in planning and problem solving. In general, learning in problem solving seemed meaningless without measurable performance improvements. Thus, PRODIGY was created to be a testbed for the systematic investigation of the loop between learning and performance in planning systems. As a result, PRODIGY consists of a core general-purpose planner and several learning modules that refine both the planning domain knowledge and the control knowledge to guide the search process effectively.

In the first design of PRODIGY, several simplifying assumptions were made in the planning algorithm, as the focus was to study how to integrate learning and planning. Thus, the first planner in PRODIGY, PRODIGY2.0, assumed linear subgoaling decomposition, i.e., no interleaving of subplans (Minton et al., 1989). The first learning technique developed was explanation-based learning of control knowledge to guide the search process. This first version of the architecture led Minton to complete an impressive thesis with an extensive empirical analysis of the effect of learning control knowledge in the planner's performance (Minton, 1988a).

After Minton's successful and pioneering results on incorporating learning into the planning framework, we pursued the investigation of alternative learning techniques in PRODIGY to address domains and problems of increasing complexity. This led to the need to evolve the PRODIGY's planning algorithm from the simple linear and incomplete PRODIGY2.0 to the current nonlinear and complete PRODIGY4.0.

The current PRODIGY architecture encompasses several learning methods that improve the performance of the core planner along several different dimensions.

The article is organized in six sections. Section 2 presents the planner, PRODIGY4.0. Section 3 describes the learning opportunities that we have addressed in PRODIGY and briefly reviews all of the learning methods of PRODIGY. To illustrate concretely how learning is integrated with planning in PRODIGY, Section 4 presents the techniques to learn to improve the quality of the plans produced by the planner. Section 5 discusses some of the characteristics of the PRODIGY architecture, shows the evolution of the system in the pursuit of our goal of building a large and robust planning and learning system, and takes a retrospective look at the project. Finally Section 6 draws conclusions on the article.

2 The PRODIGY Planning Algorithm

A planning domain is specified to the PRODIGY's planner mainly as a set of *operators*. Each operator corresponds to a generalized atomic planning action, described in terms of its effects and the necessary conditions which enable the application of the operator. A planning problem in a domain is presented to PRODIGY as an initial configuration of the world and a goal statement to be achieved.

The current PRODIGY4.0 system uses a complete nonlinear planner. It follows a means-ends analysis backward chaining search procedure reasoning about multiple goals and multiple alternative operators relevant to achieving the goals. PRODIGY4.0's nonlinear character stems from its dynamic goal selection which enables the planner to fully interleave plans, exploiting common subgoals and addressing issues of resource contention. Operators can be organized in different levels of abstractions that are used by PRODIGY4.0 to plan hierarchically.

In this section, we describe how PRODIGY4.0 combines state-space search corresponding to a simulation of plan execution of the plan and backward-chaining responsible for goal-directed reasoning. We present a formal description of PRODIGY4.0 planning algorithm. We describe and illustrate through a simple example, the planning domain language used in PRODIGY. We show how the planner represents incomplete plans considered during the search for a solution. We presents the technique to change the internal planning state to simulate the effects of plan execution, and introduce the backward-chaining algorithm. Finally, we discuss the use of control rules in PRODIGY to guide the search for a solution.

2.1 Domain and Problem Definition

A *planning domain* is defined by a set of types of objects, i.e., classes, used in the domain, a library of operators and inference rules that act on these objects. Inference rules have the same syntax as operators, hence we do not distinguish them in this section.[1] PRODIGY's language for describing operators is based on the STRIPS domain language (Fikes and Nilsson, 1971), extended to express disjunctive and negated preconditions, universal and existential quantification, and conditional effects (Carbonell et al., 1992).

Each operator is defined by its *preconditions* and *effects*. The description of preconditions and effects of an operator can contain typed variables. Variables can be constrained by functions. The *preconditions* of an operator are represented as a logical expression containing conjunctions, disjunctions, negations, universal and existential quantifiers. The *effects* of an operator consist of a set of *regular* effects, i.e., a list of predicates to be added or deleted from the state when the operator applies, and a set of *conditional* effects that are to be performed depending on particular state conditions.

When an operator is used in planning, its variables are replaced with specific objects of corresponding types. We say that the operator is *instantiated*. The precondition expression and the effects of an instantiated operator are then described in terms of *literals*, where the term *literal* refers to a predicate whose variables are instantiated with specific objects. In order to apply an instantiated operator, i.e., to execute it in the internal state of the planner, its preconditions must be satisfied in the state. Its effects specify the set of literals that must be added and deleted to the state.

Figure 1 shows an example of a planning domain, a simplified version of a compex Process Planning domain introduced in (Gil, 1991). (In Section 4, we use examples from the complete version of this domain to describe techniques for learning control rules.) This simplified domain represents operations of a drill press. The drill press uses two types of drill bits: a spot drill and a twist drill. A spot drill is used to make a small spot on the surface of a part. This spot is needed to guide the movement of a twist drill, which is used for drilling a deeper hole.

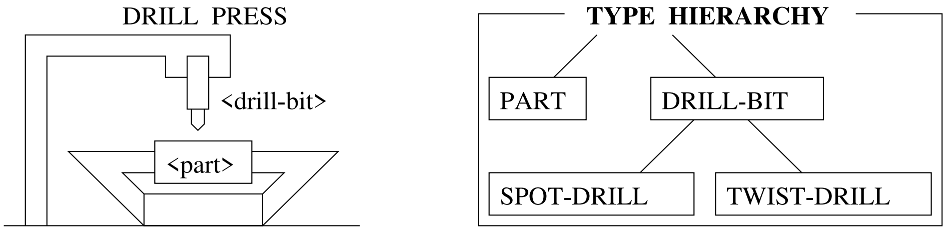
There are two types of objects in this domain: `PART` and `DRILL-BIT`. The latter type is subdivided into two subtypes, `SPOT-DRILL` and `TWIST-DRILL`. The use of types combined with functions limits the scopes of possible values of variables in the description of operators.

A *planning problem* is defined by (1) a set of available objects of each type, (2) an *initial state* I, and (3) a *goal statement* G. The initial state is represented as a set of literals. The goal statement is a logical formula equivalent to a preconditions expression, i.e. it can contain typed variables, conjunctions, negations, disjunctions, and universal and existential quantifications. An example of a planning problem is shown in Figure 2. In this example, we have five objects: two metal parts, called `part-1` and `part-2`, the spot drill `drill-1`, and two twist drills, `drill-2` and `drill-3`.

A solution to a planning problem is a sequence of operators that can be applied to the initial state, transforming it into a state that satisfies the goal. A sequence of operators is called a *total-order plan*. A plan is *valid* if the preconditions of every operator are satisfied before the execution of the operator. A valid plan that achieves the goal statement G is called *correct*.

For example, for the problem in Figure 2, the plan "`put-part(part-1),put-drill-bit(drill-1)`" is valid, since it can be executed from the initial state. However, this plan does not achieve the goal, and hence it is *not* correct. The problem may be solved by the following correct plan:

```
put-part(part-1), put-drill-bit(drill-1), drill-spot(part-1, drill-1),
remove-drill-bit(drill-1), put-drill-bit(drill-2), drill-hole(part-1, drill-2)
```



drill-spot (<part>, <drill-bit>)
<part>: type PART
<drill-bit>: type SPOT-DRILL
Pre: (holding-tool <drill-bit>)
(holding-part <part>)
Add: (has-spot <part>)

put-drill-bit (<drill-bit>)
<drill-bit>: type DRILL-BIT
Pre: tool-holder-empty
Add: (holding-tool <drill-bit>)
Del: tool-holder-empty

put-part(<part>)
<part>: type PART
Pre: part-holder-empty
Add: (holding-part <drill-bit>)
Del: part-holder-empty

drill-hole(<part>, <drill-bit>)
<part>: type PART
<drill-bit>: type TWIST-DRILL
Pre: (has-spot <part>)
(holding-tool <drill-bit>)
(holding-part <part>)
Add: (has-hole <part>)

remove-drill-bit(<drill-bit>)
<drill-bit>: type DRILL-BIT
Pre: (holding-tool <drill-bit>)
Add: tool-holder-empty
Del: (holding-tool <drill-bit>)

remove-part(<part>)
<part>: type PART
Pre: (holding-part <drill-bit>)
Add: part-holder-empty
Del: (holding-part <drill-bit>)

Figure 1: A simplified version of the Process Planning domain.

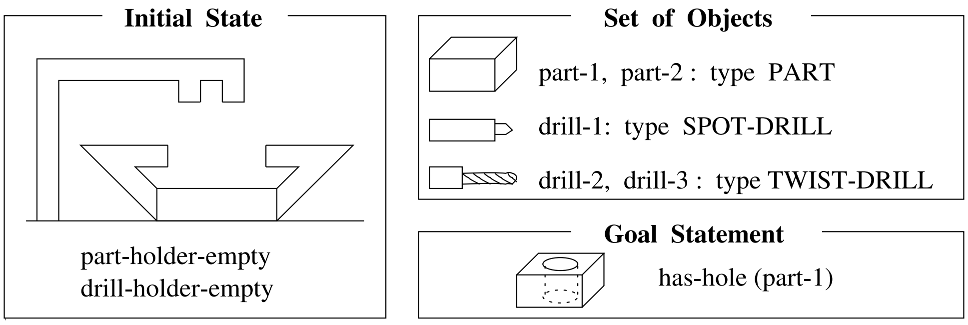


Figure 2: A problem in the simplified Process Planning domain.

A *partial-order plan* is a partially ordered set of operators. A *linearization* of a partial-order plan is a total order of the operators consistent with the plan's partial order. A partial-order plan is *correct* if all its linearizations are correct. For example, the first two operators in our solution sequence need not be ordered with respect to each other, thus giving rise to the partial-order plan in Figure 3.

Figure 3: First operators in the example partial orden plan.

2.2 Representation of Plans

Given a problem, most planning algorithms start with the empty plan and modify it until a solution plan is found. A plan may be modified by inserting a new operator, imposing a constraint on the order of operators in the plan, or instantiating a variable in the description of an operator. The plans considered during the search for a solution are called *incomplete* plans. Each incomplete plan may be viewed as a node in the search space of the planning algorithm. Modifying a current incomplete plan corresponds to expanding a node. The branching factor of search is determined by the number of possible modifications of the current plan.

Different planning systems use different ways of representing incomplete plans. A plan may be presented as a totally ordered sequence of operators (as in Strips (Fikes and Nilsson, 1971)) or as a partially ordered set of operators (as in Tweak (Chapman, 1987), NONLIN (Tate, 1977), and SNLP (McAllester and Rosenblitt, 1991)); the operators of the plan may be instantiated (e.g. in NOLIMIT (Veloso, 1989)) or contain variables with codesignations (e.g. in Tweak); the relations between operators and the goals they establish may be marked by causal links (e.g. in NONLIN and SNLP).

In PRODIGY, an incomplete plan consists of two parts, the *head-plan* and the *tail-plan* (see Figure 4). The tail-plan is built by a partial-order backward-chaining algorithm, which starts from the goal statement G and adds operators, one by one, to achieve preconditions of other operators that are untrue in the current state, i.e. *pending goals*.

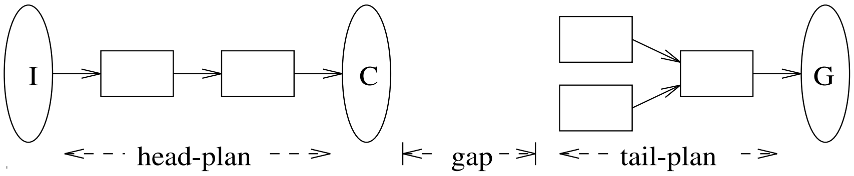


Figure 4: Representation of an incomplete plan.

The head-plan is a *valid total-order plan*, that is, a sequence of operators that can be applied to the initial state I. All variables in the operators of the head-plan are instantiated, that is, replaced with specific constants. The head-plan is generated by the execution-simulating algorithm described in the next subsection. If the current incomplete plan is successfully modified to a correct solution of the problem, the current head-plan will become the beginning of this solution.

The state C achieved by applying the head-plan to the initial state is called the *current state*. Notice that since the head-plan is a total-order plan that does not contain variables, the current state is *uniquely defined*. The back-chaining algorithm responsible for the tail-plan views C as its initial state. If the tail-plan cannot be validly executed from the current state C ,then there is a "gap" between the head and tail. The purpose of planning is to bridge this gap.

Figure 5 shows an example of an incomplete plan. This plan can be constructed by PRODIGY while solving the problem of making a spot in `part-1`. The gap in this plan can be bridged by a single operator, `put-drill-bit(drill-1)`.

2.3 Simulating Plan Execution - State Space Search

Given an initial state I and a goal statement G, PRODIGY starts with the empty plan and modifies it, step by step, until a correct solution plan is found. The empty plan is the root node in PRODIGY's search space. The head and tail of this plan are, naturally, empty, and the current state is the same as the initial state, $C = I$.

At each step, PRODIGY can modify a current incomplete plan in one of two ways (see Figure 6). It can add an operator to the tail-plan (operator *t* in Figure 6). Modifications of the tail are handled by a backward-chaining planning algorithm, called *Back-Chainer*, which is presented in the next subsection. PRODIGY can also move some operator *op* from the tail to the head (operator *x* in Figure 6). The preconditions of *op* must

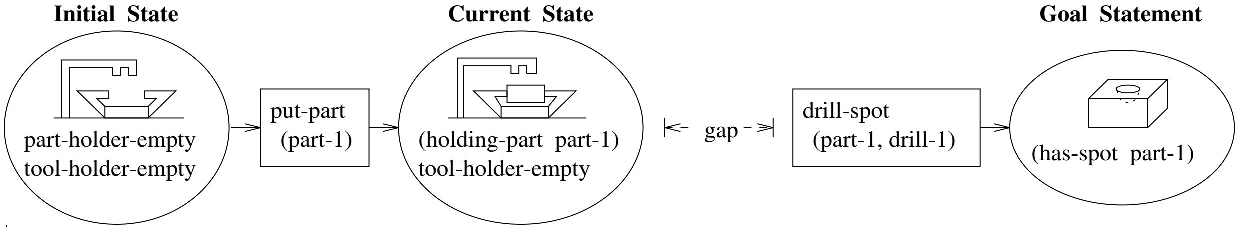


Figure 5: Example of an incomplete plan.

be satisfied in the current state C. *op* becomes the last operator of the head, and the current state is updated to account for the effects of *op*. There may be several operators that can be applied to the state. There are usually also many possible goals to plan for, and operators and instantiations of operators that can achieve each particular goal. These choices are made while expanding the tail-plan.

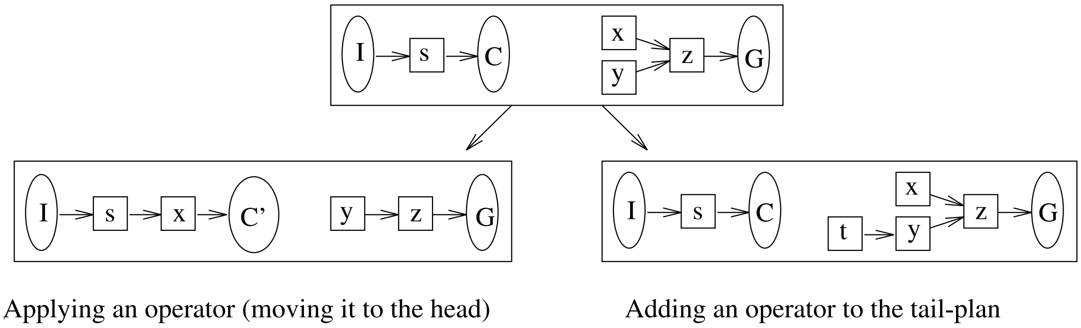


Figure 6: Modifying the current plan.

Intuitively, one may imagine that the head-plan is being carried out in the real world, and PRODIGY has already changed the world from its initial state I to the current state C. If the tail-plan contains an operator whose preconditions are satisfied in C, PRODIGY can apply it, thus changing the world to a new state, say C'. Because of this analogy with the real-world changes, the operation of moving an operator from the tail to the end of the head is called the *application* of an operator. Notice that the term "application" refers to *simulating* an operator application. Even if the application of the current head-plan is disastrous, the world does not suffer: PRODIGY simply backtracks and considers an alternative execution sequence.

Moving an operator from the tail to the head is one way of updating the head-plan. PRODIGY can perform additional changes to the state adding redundant explicit information by inference rules.

PRODIGY recognizes a plan as a solution of the problem if the head-plan achieves the goal statement G, i.e. the goal statement is satisfied in C. PRODIGY may terminate after finding a solution, or it may search for another plan.

Table 1 summarizes the execution-simulating algorithm. The places where PRODIGY chooses among several alternative modifications of the current plan are marked as *decision* points. To guarantee completeness, PRODIGY must consider all alternatives at every decision point.

Notice that, by construction, the head-plan is always valid. PRODIGY terminates when the goal statement G is satisfied in the current state C, achieved by applying the head-plan. Therefore, upon termination, the head-plan is always valid and achieves G, that is, it is a correct solution of the planning problem. We conclude that PRODIGY is a sound planner, and its soundness does not depend on *Back-Chainer*.

Prodigy
1. If the goal statement G is satisfied in the current state C, then return *Head-Plan*.
2. Either
   (A) *Back-Chainer* adds an operator to the *Tail-Plan*, or
   (B) *Operator-Application* moves an operator from *Tail-Plan* to *Head-Plan*.
   Decision point: Decide whether to apply an operator or to add an operator to the tail.
3. Recursively call *Prodigy* on the resulting plan.

Operator-Application
1. Pick an operator *op* in *Tail-Plan* which is an *applicable operator*, that is
   (A) there is no operator in *Tail-Plan* ordered before *op*, and
   (B) the preconditions of *op* are satisfied in the current state C.
   Decision point: Choose an applicable operator to apply.
2. Move *op* to the end of *Head-Plan* and update the current state C.

Table 1: Execution-simulating algorithm.

2.4 Backward-Chaining Algorithm

We now turn our attention to the backward-chaining technique used to construct PRODIGY's tail-plan. The tail-plan is organized as a tree (see Figure 7). The root of the tree is a ficticious operator, **finish**, that adds the goal statement G, the other nodes are operators, and the edges are ordering constraints. Each operator of the tail-plan is added to the tail-plan to achieve a pending goal. Every operator gets instantiated immediately after it is inserted into the tail[2] .

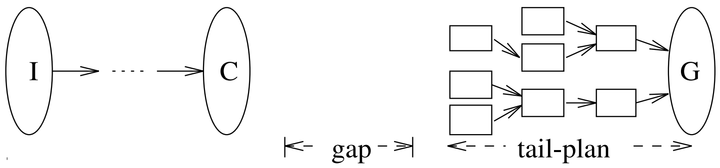


Figure 7: Tree-structured tail-plan.

A precondition literal l in a tree-like tail-plan is considered *unachieved* if
(1) l does not hold in the current state C, and
(2) l is not linked with any operator of the tail-plan.
When the backward-chaining algorithm is called to modify the tail, it chooses some unachieved literal l, adds a new operator *op* that achieves l, and establishes an ordering constraint between *op1* and *op2*. *op2* is also marked as the "relevant operator" to achieve *l*.

Before inserting the operator *op* into the tail-plan, *Back-Chainer* substitutes all free variables of *op*. Since PRODIGY's domain language allows complex constraints on the preconditions of operators, finding the set of possible substitutions may be a difficult problem. This problem is handled by a constraint-based matching algorithm, described in (Wang, 1992).

For example, suppose the planner adds the operator `drill-hole(<part>,<drill-bit>)` to the tail-plan in order to achieve the goal literal `has-hole(part-1)`, as the effects of the operator unify with the goal (see Figure 8). First, PRODIGY instantiates the variable <`part`> in the description of the operator with the constant `part-1` from the goal literal. Then the system has to instantiate the remaining free variable, `<drill-bit>`. Since our problem contains two twist drills, `drill-2` and `drill-3`, this variable has two possible instantiations. Different instantiations give rise to different branches of the search space.

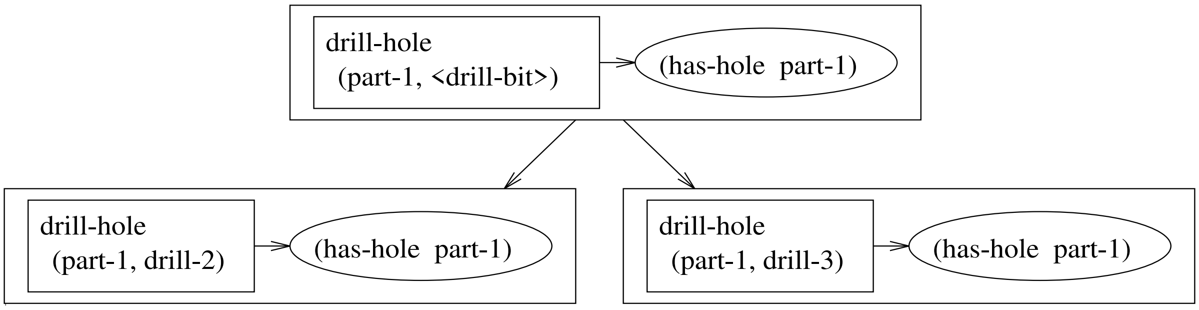


Figure 8: Instantiating a newly added operator.

A summary of the backward-chaining algorithm is presented in Table 2. It may be shown that the use of this back-chaining algorithm with the execution-simulator described in the previous subsection guarantees the completeness of planning (Fink and Veloso, 1994). Notice that the decision point in Step 1 of the algorithm may be required for the efficiency of the depth-first planning, but not for completeness. If PRODIGY generates the entire tail before applying it, we do not need branching on Step 1. We may pick preconditions in any order, since it does not matter in which order we add nodes to our tail-plan.

Back-Chainer

1. Pick an unachieved goal or precondition literal l.
   Decision point: Choose an unachieved literal.
2. Pick an operator *op* that achieves l.
   Decision point: Choose an operator that achieves this literal.
3. Add *op* to the plan and establish a link from *op* to l.
4. Instantiate the free variables of *op*.
   Decision point: Choose an instantiation for the variables of the operator.

Table 2: Backward-chaining algorithm.

2.5 Control Rules

The algorithms presented in the previous section (Tables 1 and 2) determine the search space of the PRODIGY planner. Different choices in decision points of the algorithms give rise to different ways of exploring the search space. Usually PRODIGY uses depth-first search to explore its search space. The efficiency of the depth-first search crucially depends on the choices made at decision points. Figure 9 summarizes the decisions made by the system at every step of planning which are directly related to the choice points in Tables 1 and 2.

Strategies used in PRODIGY for directing its choices in decision points are called *control knowledge*. These strategies include the use of control rules (usually domain-dependent) (Minton, 1988a), complete problem solving episodes to be used by analogy (Veloso and Carbonell, 1993), and domain-independent heuristics (Blythe and Veloso, 1992; Stone et al., 1994). We focus on explaining control rules.

A *control rule* is a production (*if-then* rule) that tells the system which choices should be made (or avoided) depending on the current state C, unachieved preconditions of the tail-plan operators, and other meta-level information based on previous choices or subgoaling links. Control rules can be hand-coded by the user or automatically learned by the system. Some techniques for learning control rules are presented in (Minton, 1988a), (Etzioni, 1990) and (Pérez and Etzioni, 1992). In this paper we will present two more rule-learning techniques (see Section 4. Figure 9 shows the decisions made by PRODIGY, that are also the decisions that can be controlled by control rules. This figure can be very directly compared with Figure 6. We see that a search node corresponds to a sequence of decisions which will be the learning opportunities.

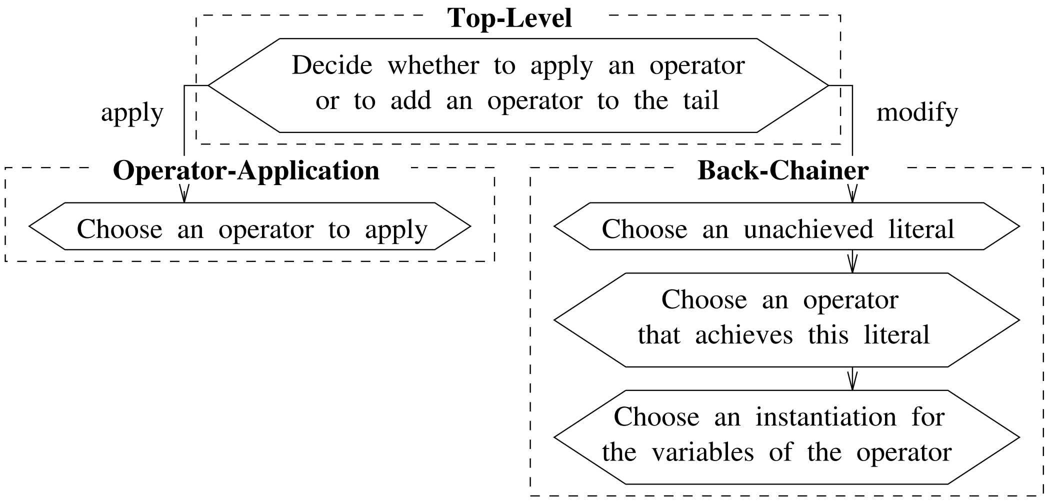


Figure 9: Branching decisions.

Two examples of control rules are shown in Table 3. The first rule says that if we have just inserted an operator `drill-hole(<part>, <drill>)` to achieve an unsatisfied goal literal `(has-hole <drill>)`, and if the literal `(holding-drill-bit <drill-1>)` is satisfied in the current state for some specific drill `<drill-1>`, then we should instantiate the variable `<drill>` with `<drill-1>`, and we should *not* consider any other instantiations of this variable. This is equivalent to saying that all the drills are identical from this perspective. Thus, this rule selects one of the alternatives and prunes all other branches from the search space. Such a rule is called a *select* rule. Similarly, we may define a control rule that points out an alternative that should *not* be considered, thus pruning some specific branch of the search space. The rules of this type are called *reject* rules.

The second rule in Table 3 says that if we have two unsatisfied goal literals, `(has-hole <part-1>)` and `(has-hole <part-2>)`, and if `<part-1>` is already in the drill press, then the first goal must be considered *before* the second one. Thus, this rule suggests that some branch of the search space must be explored before some other branch. The rules that provide knowledge on the preferred order of considering different alternatives are called *prefer* rules.

All control rules used by PRODIGY are divided into these three groups: *select*, *reject*, and *prefer* rules. If there are several control rules applicable in the current decision point, PRODIGY will use all of them. *Select* and *reject* rules are used to prune parts of the search space, while *prefer* rules determine the order of exploring the remaining parts.

First, PRODIGY applies all *select* rules whose *if*-part matches the current situation, and creates a set of candidate branches of the search space that must be considered. A branch becomes a candidate if at least one

Select Rule

If (has-spot <part>) is the current goal
and drill-spot (<part>, <drill>) is the current operator
and (holding-drill-bit <drill-1>) holds in the current state
and <drill-1> is of the type SPOT-DRILL
Then select instantianting <drill> with <drill-1>

Prefer Rule

If (has-hole <part-1>) is a candidate goal
and (has-hole <part-2>) is a candidate goal
and (holding-part <part-1>) holds in the current state
Then prefer the goal (has-hole <part-1>) to (has-hole <part-2>)

Table 3: Examples of control rules.

select rule points to this branch. If there are no *select* rules applicable in the current decision point, then by default *all* branches are considered candidates. Next, PRODIGY applies all *reject* rules that match the current situation and prunes every candidate pointed to by at least one *reject* rule. Notice that an improper use of *select* and *reject* rules may violate the completeness of planning, since these rules can prune a solution from the search space. They should therefore be used carefully.

After using *select* and *reject* rules to prune branches of the search space, PRODIGY applies *prefer* control rules to determine the order of exploring the remaining candidate branches. When the system does not have applicable *prefer* rules, it may use general domain-independent heuristics for deciding on the order of exploring the candidate branches. These heuristics are also used if the applicable *prefer* rules contradict each other in ordering the candidate branches.

When PRODIGY encounters a new planning domain, it initially relies on the control rules specified by the user (if any) and domain-independent heuristics to make branching decisions. Then, as PRODIGY learns control rules and stores problem-solving episodes for analogical case-based reasoning, the domain-independent heuristics are gradually overriden by the learned control knowledge.

3 Learning in PRODIGY

The PRODIGY planning algorithm is combined with several learning modules, designed for reducing the planning time, improving the quality of the solution plans, and refining the domain knowledge. Figure 10 shows the learning modules developed in PRODIGY, according to their learning goal.

We describe briefly some of these algorithms. In the following section, we present in more detail two of PRODIGY's learning modules to improve the quality of the plans produced by the planner.

3.1 Learning Control Knowledge to Improve Planning Efficiency

The following are PRODIGY's learning modules to acquire control knowledge to improve planning efficiency:

EBL: An explanation-based learning facility (Minton, 1988a) for acquiring control rules from a problem-solving trace. EBL analyzes the search space of the planning algorithm and explains the reasons for branching decisions made during the search for a solution. The resulting explanations are expressed in the form of control rules.

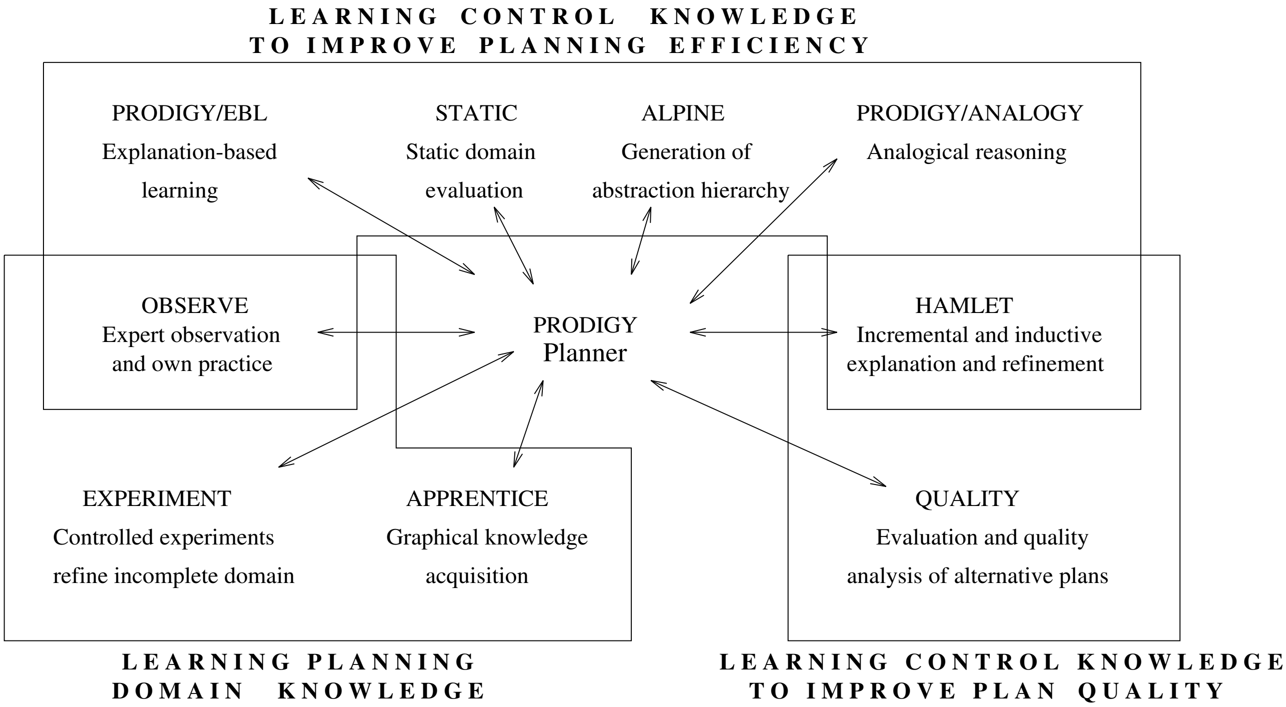


Figure 10: The learning modules in the PRODIGY architecture. We identify three classes of learning goals: learn control knowledge to improve the planner's *efficiency* in reaching a solution to a problem; learn control knowledge to improve the *quality* of the solutions produced by the planner; and learn *domain* knowledge, i.e., learn or refine the set of operators specifying the domain.

STATIC: A method for learning control rules by analyzing PRODIGY's domain description prior to planning. The STATIC program produces control rules without analizing examples of planning problems. Experiments show that STATIC runs considerably faster than EBL, and the control rules found by STATIC are usually superior to those generated by EBL. However, some rules cannot be found by STATIC and require EBL's dynamic capabilities to learn them. STATIC's design is based on a detailed predictive theory of EBL, described in (Etzioni, 1993).

DYNAMIC: A technique that combines EBL and STATIC. Training problems pinpoint learning opportunities but do not determine EBL's explanations. DYNAMIC uses the analysis algorithms introduced by STATIC but relies on training problems to achieve the distribution-sensitivity of EBL (Pérez and Etzioni, 1992).

ALPINE: An abstraction learning and planning module (Knoblock, 1994). ALPINE divides the description of a planning domain into multiple levels of abstraction. The planning algorithm uses the abstract description of the domain to find an "outline" of a solution plan and then refines this outline by adding necessary details.

ANALOGY: A derivational analogy engine that solves a problem using knowledge about similar problems solved before (Veloso and Carbonell, 1990; Veloso, 1992). The planner records the justification for each branching decision during its search for a solution. Later, these justifications are used to guide the search for solutions of other similar problems.

3.2 Learning Domain Knowledge

Acquiring planning knowledge from experts is a rather difficult knowledge engineering task. The following learning modules support the user with the specification of domain knowledge, both at acquisition and refinement times.

EXPERIMENT: A learning-by-experimentation module for refining the domain description if the properties of the planning domain are not completely specified (Carbonell and Gil, 1990; Gil, 1992). This module is used when PRODIGY notices the difference between the domain description and the behavior of the real domain. The *experiment* module is aimed at refining the knowledge about the properties of the real world rather than learning control knowledge.

OBSERVE: This work provides a novel method for accumulating domain planning knowledge by learning from the observation of expert planning agents and from one's own practice. The observations of an agent consist of: 1) the sequence of actions being executed, 2) the state in which each action is executed, and 3) the state resulting from the execution of each action. Planning operators are learned from these observation sequences in an incremental fashion utilizing a conservative specific-to-general inductive generalization process. Operators are refined and new ones are created extending the work done in EXPERIMENT. In order to refine the new operators to make them correct and complete, the system uses the new operators to solve practice problems, analyzing and learning from the execution traces of the resulting solutions or execution failures (Wang, 1994).

APPRENTICE: A user interface that can participate in an apprentice-like dialogue, enabling the user to evaluate and guide the system's planning and learning. The interface is graphic-based and tied directly to the planning process, so that the system not only acquires domain knowledge from the user, but also uses the user's advices for guiding the search for a solution of the current problem (Joseph, 1989; Joseph, 1992).

All of the learning modules are loosely integrated: they all are used with the same planning algorithm on the same domains. The learned control knowledge is used by the planner.

We are investigating ways to integrate the learning modules understanding the tradeoffs of the conditions for which each individual technique is more appropriate. Learning modules will then share the generated knowledge with each other and a tighter integration will be achieved. As an initial effort in this direction, EBL was used in combination with the abstraction generator to learn control rules for each level of abstraction. The integration of these two modules in PRODIGY2.0 simplifies the learning process and results in generating more general control rules, since the proofs in an abstract space contain fewer details (Knoblock et al., 1991).

4 Learning to Improve Plan Quality

Most of the research on planning so far has concentrated on methods for constructing sound and complete planners that find a satisficing solution, and on how to find such a solution efficiently (Chapman, 1987; McAllester and Rosenblitt, 1991; Peot and Smith, 1993). Accordingly most of our work on integrating machine learning and planning, as described above, has focused on learning control knowledge to improve the planning efficiency and learning to acquire or refine domain knowledge. Recently, we have developed techniques to learn to improve the *quality of the plans* produced by the planner. Generating production-quality plans is an essential element in transforming planners from research tools into real-world applications.

4.1 Plan Quality and Planning Efficiency

Planning goals rarely occur in isolation and the interactions between conjunctive goals have an effect in the quality of the plans that solve them. In (Pérez and Veloso, 1993) we argued for a distinction between *explicit goal interactions* and *quality goal interactions*. Explicit goal interactions are represented as part of the domain knowledge in terms of preconditions and effects of the operators. Given two goals g_1 and g_2 achieved respectively by operators op_1 and op_2, if op_1 deletes g_2 then goals g_1 and g_2 interact because there is a strict ordering between op_1 and op_2. On the other hand, quality goal interactions are not directly related to successes and failures. As a particular problem may have many different solutions, quality goal interactions may arise as the result of the particular problem solving search path explored. For example, in a machine-shop scheduling domain, when two identical machines are available to achieve two goals, these goals may interact, if the problem solver chooses to use just one machine to achieve both goals, as it will have to wait for the machine to be idle. If the problem solver uses the two machines instead of just one, then the goals do not interact in this particular solution. These interactions are related to plan quality as the use of resources dictates the interaction between the goals. Whether one alternative is better than the other depends on the particular quality measure used for the domain. The control knowledge to guide the planner to solve these interactions is harder to learn automatically, as the domain theory, i.e. the set of operators, does not encode these quality criteria.

It can be argued that the heuristics to guide the planner towards better solutions could be incorporated into the planner itself, as in the case of removing unnecessary steps at the end of planing, or using breadth-first search when plan length is used as the evaluation function. However in some domains plan length is not an accurate metric of plan quality, as different operators have different costs. More sophisticated search heuristics should then be needed. The goal of learning plan quality is to automatically acquire heuristics that guide the planner at generation time to produce plans of good quality.

There are several ways to measure plan quality as discussed in (Pérez and Carbonell, 1993). Our work on learning quality-enhancing control knowledge focuses on quality metrics that are related to plan execution cost expressed as an linear evaluation function additive on the cost of the individual operators.

In this section we present two strategies for attacking this learning problem, implemented respectively in two learning modules, QUALITY and HAMLET.

QUALITY assumes that different final plans can be evaluated by an evaluation function to identify which of several alternative plans is the one of better quality. QUALITY provides a plan checker to allow a user to interactively find valid variations of a plan produced. The system then analyzes the differences between the sequence of decisions that the planner encountered and the ones that could have been selected to generate a plan of better quality. The learner interprets these differences and identifies the conditions under which the individual planning choices will lead to the desired final plan. QUALITY compiles knowledge to control the decision making process in new planning situations to generate plans of better quality.

HAMLET[3] acquires control knowledge to guide PRODIGY to efficiently produce cost-effective plans. HAMLET learns from exhaustive explorations of the search space in simple problems, loosely explains the conditions of quality success and refines its learned knowledge with experience.

4.2 QUALITY: Learning by Explaining Quality Differences

Knowledge about plan quality is domain dependent but not encoded in the domain definition, i.e., in the set of operators and inference rules, and might vary over time. To capture such knowledge, QUALITY assumes that the planner's domain theory is extended with an function that evaluates the quality of final produced plans. Learning consists in a *translation* task of the plan evaluation function into control knowledge usable by the planner at plan generation time. Figure 11 shows the architecture of QUALITY implemented within PRODIGY4.0.

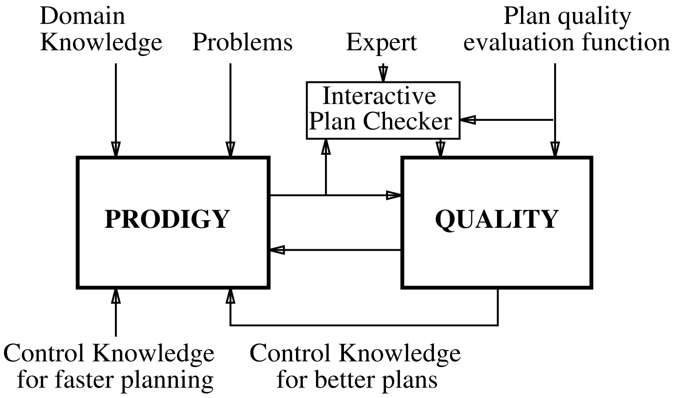


Figure 11: Architecture of the learning module QUALITY, to learn control knowledge to improve the quality of plans.

The learning algorithm is *given* a domain theory (operators and inference rules) and a domain-dependent function that evaluates the quality of the plans produced. It is also given problems to solve in that domain. QUALITY analyzes the problem-solving episodes by comparing the search trace for the planner solution given the current control knowledge, and another search trace corresponding to a better solution (*better* according to the evaluation function). The latter search trace is obtained by asking a human expert for a better solution and then producing a search trace that leads to that solution, or by letting the problem solver search further until a better solution is found. The algorithm explains why one solution is better than the other and *outputs* search control knowledge that directs future problem solving towards better quality plans. Two points are worth mentioning:

- Learning is driven by the existence of a better solution and a failure of the current control knowledge to produce it.
- There is a change of representation from the knowledge about quality encoded in the domain-dependent plan evaluation function into knowledge operational at planning time, as the plan and search tree are only partially available when a decision has to be made. The translated knowledge is expressed as search-control knowledge in terms of the problem solving state and meta-state, or tail plan, such as which operators and bindings have been chosen to achieve the goals or which are the candidate goals to expand.

We do not claim that this control knowledge will necessarily guide the planner to find optimal solutions, but that the quality of the plans will incrementally improve with experience, as the planner sees new interesting problems in the domain.

4.2.1 QUALITY: The Algorithm

Table 4 shows the basic procedure to learn quality-enhancing control knowledge, in the case that a human expert provides a better plan. Steps 2, 3 and 4 correspond to the interactive plan checking module, that asks the expert for a better solution S_e and checks for its correctness. Step 6 constructs a problem solving trace from the expert solution and obtains decision points where control knowledge is needed, which in turn become learning opportunities. Step 8 corresponds to the actual learning phase. It compares the plan trees obtained from the problem solving traces in Step 7, explains why one solution was better than the other, and builds new control knowledge. These steps are described now in detail.

1. Run PRODIGY with the current set of control rules and obtain a solution S_p.
2. Show S_p to the expert.
   Expert provides new solution S_e possibly using S_p as a guide.
3. Test S_e. If it solves the problem, continue. Else go back to step 2.
4. Apply the plan quality evaluation function to S_e.
   If it is better than S_p, continue. Else go back to step 2.
5. Compute the partial order $\mathcal{P}$ for S_e identifying the goal dependencies between plan steps.
6. Construct a problem solving trace corresponding to a solution S'_e that satisfies $\mathcal{P}$.
   This determines the set of decision points in the problem solving trace where control knowledge is missing.
7. Build the plan trees T'_e and T_p, corresponding respectively to the search trees for S'_e and S_p.
8. Compare T'_e and T_p explaining why S'_e is better than S_p, and build control rules.

Table 4: Top level procedure to learn quality-enhancing control knowledge.

Interactive plan checker QUALITY interacts with an expert to determine variations to the plan currently generated that may produce a plan of better quality. We have built an interactive plan checker to obtain a better plan from the domain expert and test whether that plan is correct and actually better. The purpose of the interactive plan checker is to capture the expert knowledge about how to generate better plans in order to learn quality-enhancing control knowledge. The interaction with the expert is at the level of plan steps, i.e. instantiated operators, and not of problem-solving time decisions, relieving the expert of understanding PRODIGY's search procedure.

The plan checker offers the expert the plan obtained with the current knowledge as a guide to build a better plan. It is interactive as it asks for and checks one plan step at a time. The expert may pick an operator from the old plan, which is presented in menu form, or propose a totally new operator. There is an option also to see the current state. The plan checker verifies the correctness of the input operator name and of the variable bindings according to the type specifications and may suggest a default value if some is missing or wrong. If the operator is not applicable in the current state, the checker notifies the expert which precondition failed to be true. When the new plan is completed, if the goal is not satisfied in the final state, or the plan is worse than the previous plan according to the plan quality evaluation function, the plan checker informs and prompts the expert for another plan, as the focus is on learning control knowledge that actually improves solution quality.

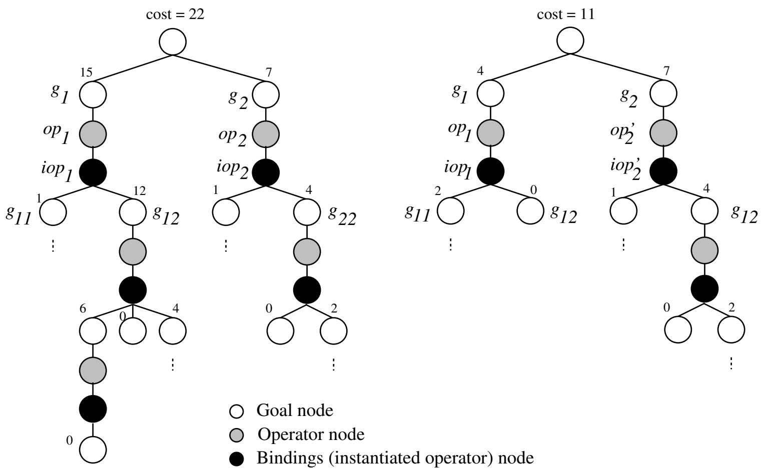
As we mentioned, the interaction with the user is at the level of operators, which represent actions in the world. In domains with inference rules, they usually do not correspond semantically to actions in the world and are used only to compute the deductive closure of the current state. Therefore the plan checker does not require that the user specifies them as part of the plan, but it fires the rules needed in a similar way to the problem solver. A truth maintenance system keeps track of the rules that are fired, and when an operator is applied and the state changes, the effects of the inference rules whose preconditions are no longer true are undone.

Constructing a problem solving trace from the plan Once the expert has provided a good plan, the problem solver is called in order to generate a problem solving trace that corresponds to that plan. However we do not require that the solution be precisely the same. Because our plan quality evaluation functions are additive on the cost of operators, any linearization of the partial order corresponding to the expert solution has the same cost, and we are content with obtaining a trace for any of them. (Veloso, 1989) describes the algorithm that generates a partial order from a total order.

To force the problem solver to generate a given solution, we use PRODIGY's interrupt mechanism (Carbonell et al., 1992), which can run user-provided code at regular intervals during problem solving. At each decision node it checks whether the current alternative may be part of the desired solution and backtracks otherwise. For example, at a bindings node it checks whether the instantiated operator is part of the expert's plan, and if not it tries a different set of bindings. The result of this process is a search trace that produces the expert's solution, and a set of decision points where the current control knowledge (or lack of it) was overridden to guide PRODIGY towards the expert's solution. Those are learning opportunities for our algorithm.

Building plan trees At this point, two problem solving traces are available, one using the current control knowledge, and other corresponding to the better plan. QUALITY builds a *plan tree* from each of them. The nodes of a plan tree are the goals, operators, and bindings, or instantiated operators, considered by PRODIGY during problem solving in the successful search path that lead to that solution. In this tree, a goal is linked to the operator considered to achieve the goal, the operator is linked in turn to its particular instantiation, i.e. bindings, chosen and the bindings are linked to the subgoals corresponding to the instantiated operator's preconditions. Leaf nodes correspond to subgoals that were true when PRODIGY tried to achieve them. The reason for them being true is stored in the goal node: either they were true in the initial state, or were achieved by some operator that was chosen for other subgoal. Figure 12 shows the plan trees corresponding to two plans for the same problem.

Next the plan quality evaluation function is used to assign *costs* to the nodes of the plan tree, starting with the leaves and propagating them back up to the root. The leaf nodes have cost 0. An operator node has the cost given by the evaluation function, plus the sum of the costs of achieving its preconditions, i.e the costs of its children subgoals. Operator and bindings nodes have the same cost. A goal node has the cost of the operator used to achieve it, i.e. of its child. Note that if a goal had to be reachieved, it has more than one child operator. Then the cost of achieving the goal is the sum of the costs of the children operators. The root cost will be the cost of the plan. The plans for the plan trees in Figure 12 have different costs, with the tree on the left corresponding to the plan of worse quality. Note that the solutions differ in the operator and bindings chosen to achieve g_2. Those decisions are the ones for which the guidance provided by control knowledge is needed.



e_{iop}: cost of instantiated operator iop, given by the plan quality evaluation function
$c(n)$: cost associated with node n

$\forall$ goal node n_g,
$c(n_g) = 0$ if n_g is a leaf node
$c(n_{op})$ if n_{op} is a child of n_g
$\forall$ op node n_{op},
$c(n_{op}) = c(n_{iop})$ if n_{iop} is a child of n_{op}
$\forall$ bindings node n_{iop},
$c(n_{iop}) = e_{iop} + \sum_{n_g \in children(n_{iop})} c(n_g)$

Figure 12: Plan trees corresponding to two solutions of different quality for the same problem. The cumulative cost of each subtree is propagated up to its ancestrors. A number next to a node indicates the cost of the subtree rooted at that node.

Learning control rules Using the two plan trees, the algorithm explains why one solution is better than the other. The cost of the two plan trees is compared recursively, starting with the root, looking for cost differences. In the trees of Figure 12 the cost of achieving g_2 is the same for both trees, but the cost of achieving g_1 is different. Therefore g_1's subtrees are explored in turn. The cost of the children operators is the same, as they are the same operators. However one of their subgoals, g_{12}, was less expensive to achieve in the better solution. In fact it had cost zero because it was shared with the subtree corresponding to g_2. The algorithm considers that this explains the different costs of the plans. Note that it ignores subgoals that had higher cost in the cheaper solution, such as g_{11}, therefore introducing incompleteness in the explanation construction.

This explanation, namely that g_{12} is shared, is made operational, i.e. expressed using knowledge available to the planner at decision time. Step 6 in the algorithm (Table 4) indicated the learning opportunities, or wrong decisions made by the planner where new control knowledge was required. op_2' with bindings corresponding to iop_2' should be preferred over op_2. This corresponds to an operator and bindings decision. The explanation is propagated up the subtrees until it reaches the node corresponding to the decision point, gathering the conditions under which g_{12} is a shared subgoal. At that node g_{12} was a pending goal (a subgoal of op_1), and for g_{12} to be a shared subgoal, precisely op_2' should be chosen with the appropriate set of bindings in iop_2'. With these facts the control rules are built. The learning opportunity defines the right-hand side of each rule, i.e. which operator or bindings to prefer, and some of the preconditions such as what the current goal is. The rest of the preconditions come from the explanation. The algorithm is also able to learn goal preferences.

4.2.2 Example: Plan Quality in the Process Planning Domain

To illustrate the algorithm just described, we introduce an example in the domain of process planning for production manufacturing. In this domain plan quality is crucial in order to minimize both resource consumption and execution time. The goal of process planning is to produce plans for machining parts given their specifications. Such planning requires taking into account both technological and economical considerations (Descotte and Latombe, 1985; Doyle, 1969), for instance: it may be advantageous to execute several cuts on the same machine with the same fixing to reduce the time spent setting up the work on the machines; or, if a hole H_1 opens into another hole H_2, then one is recommended machining H_2 before H_1 in order to avoid the risk of damaging the drill. Most of these considerations are not pure constraints but only preferences when compromises are necessary. They often represent both the experience and the know-how of engineers, so they may differ from one company to the other.

Section 2 gave a (very) simplified example of the actions involved in drilling a hole. Here we use a much more realistic implementation of the domain in PRODIGY (Gil, 1991; Gil and Pérez, 1994). This implementation concentrates on the machining, joining, and finishing steps of production manufacturing. In order to perform an operation on a part, certain set-up actions are required as the part has to be secured to the machine table with a holding device in certain orientation, and in many cases the part has to be clean and without burrs from preceding operations. The appropriate tool has to be selected and installed in the machine as well. Table 5 shows the initial state and goal for a problem in this domain. The goal is to produce a part with a given height, namely 2, and with a spot hole on one of its sides, namely `side1`. The system also knows about a domain-dependent function that evaluates the quality of the plans and is additive on the plan operators. If the right operations and tools to machine one or more parts are selected so portions of the set-ups may be shared, one may usually reduce the total plan cost. The plan quality metric used in this example captures the different cost of setting up parts and tools on the machines required to machine the part.

Assume that, with the current control knowledge, the planner obtains the solution in Table 6 (a)

```
(state (and (diameter-of-drill-bit twist-drill6 9/64)
            (material-of part5 ALUMINUM) (size-of part5 LENGTH 5)
            (size-of part5 HEIGHT 3) (size-of part5 WIDTH 3)...))

(goal ((<part> PART)) (and (size-of <part> HEIGHT 2)
                           (has-spot <part> hole1 side1 1.375 0.25)))
```

Table 5: Example problem, showing the goal and a subset of the initial state.

that uses the drill press to make the spot hole. For example, the operation <drill-with-spot-drill drill1 spot-drill1 vise1 part5 hole1 side1 side2-side5> indicates that hole1 will be drilled in part5 on the drill press drill1 using as a tool spot-drill1, while the part is being held with its side 1 up, and sides 2 and 5 facing the holding device, vise1. This is not the only solution to the problem. In this domain milling machines, but not drill presses, can be used both to reduce the size of a part, and to drill holes and spot holes on it. Therefore there exists a better plan, according to the evaluation function, that uses the milling machine to drill the spot hole, and shares the same set-up (machine, holding device, and orientation) for the drill and mill operations, so that the part does not have to be released and held again. These differences correspond at problem-solving time to an operator decision (prefer drill-with-spot-drill-in-milling-machine over drill-with-spot-drill) and an instantiation decision (bindings for drill-with-spot-drill-in-milling-machine). Table 6 (b) shows the improved plan. This plan may be input by a human expert using the interactive plan checker. The search traces corresponding to plans (a) and (b) are transformed in plan trees, similar to the ones shown in Figure 12. If the hole is drilled in the milling-machine using the proper holding device and orientation, the subgoal (holding milling-mach1 vise1 part5 side1 side3-side6) is shared by the face-mill and drill operations. To explain that holding with those particular bindings is shared, the algorithm traverses the tree up from that goal node finding that drill-with-spot-drill-in-milling-machine should be chosen with the appropriate set of bindings, which includes using a machine of type milling machine and a particular holding device and orientation. Note that the machine type determines not only the bindings

```
put-tool-drill drill1 spot-drill1
put-holding-device-drill drill1 vise1
clean part5
put-on-machine-table drill1 part5
hold drill1 vise1 part5 side1 side2-side5
drill-with-spot-drill drill1 spot-drill1 vise1
   part5 hole1 side1 side2-side5

put-tool-mm milling-mach1 plain-mill1
release drill1 vise1 part5
remove-holding-device drill1 vise1
put-holding-device-mm milling-mach1 vise1
remove-burrs part5 brush1
clean part5
put-on-machine-table milling-mach1 part5
hold milling-mach1 vise1 part5 side1 side3-side6
face-mill milling-mach1 part5 plain-mill1 vise1
   side1 side3-side6 height 2

cost = 28
```

(a)

```
put-tool-mm milling-mach1 spot-drill1
put-holding-device-mm milling-mach1 vise1
clean part5
put-on-machine-table milling-mach1 part5
hold milling-mach1 vise1 part5 side1 side3-side6
drill-with-spot-drill-mm milling-mach1
   spot-drill1 vise1 part5 hole1 side1 side3-side6
remove-tool milling-mach1 spot-drill1
put-tool-mm milling-mach1 plain-mill1

face-mill milling-mach1 part5 plain-mill1
   vise1 side1 side3-side6 height 2

cost = 15
```

(b)

Table 6: (a) Plan obtained by PRODIGY guided by the current control knowledge. (b) A better plan, according to the evaluation function, possibly input by a human expert.

but the operator itself. The explanation is made operational to the point where the operator and bindings decisions should be made; at that point `holding` was a pending goal (a precondition of the operator `face-mill`). Table 7 shows the two control rules learned from this example. The rules indicate which operator and bindings are preferred to achieve the current goal, namely to make a spot hole in a certain part side, if a pending goal in the tail plan is to hold that part in a milling machine in certain orientation and with certain holding device.

```
(control-rule pref-drill-with-spot-drill-in-milling-machine30
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (pending-goal (holding <mach> <holding-dev> <part> <side> <side-pair>))
          (type-of-object <mach> milling-machine)))
 (then prefer operator drill-with-spot-drill-in-milling-machine
                       drill-with-spot-drill))

(control-rule pref-bnds-drill-with-spot-drill-in-milling-machine31
 (if (and (current-goal (has-spot <part> <hole> <side> <loc-x> <loc-y>))
          (current-operator drill-with-spot-drill-in-milling-machine)
          (pending-goal (holding <mach4> <holding-dev5> <part> <side> <side-pair-6>))
          (or (diff <mach4> <mach1>) (diff <holding-dev5> <holding-dev2>)
              (diff <side-pair-6> <side-pair-3>))))
 (then prefer bindings  ((<mach> . <mach4>)(<hd> . <holding-dev5>)(<sp> . <side-pair-6>))
                        ((<mach> . <mach1>)(<hd> . <holding-dev2>)(<sp> . <side-pair-3>))))
```

Table 7: Operator and bindings preference control rules learned from the problem in Table 5.

4.2.3 Discussion

QUALITY builds an explanation from a single example of a difference between a plan generated and a modification of that plan as suggested by an expert. It compiles the reasons why one alternative at the decision point is preferred over the other. Therefore the explanation is *incomplete* as it does not consider all possible hypothetical scenarios. For example, assume that a third goal in the example problem is to have `hole1` counterbored. Counterboring can only be performed by the drill press. Therefore using the milling machine for the spot hole may not be the best alternative, as the part has to be set up also in the drill press. However the rules in Table 7 would still fire choosing the milling machine, because they are too general. The consequence of using incomplete explanations is learning over-general knowledge. Our system refines the learned knowledge incrementally, upon unexpected failures. By a failure we mean that the preferred alternative leads to a plan that can be improved. The refinement does not modify the applicability conditions of the learned rules but adds new rules if needed and sets priorities among rules. In the counterboring example, a more specific rule that considers the existence of the counterboring goal is learned and given priority over the previously-learned more general rule. In the next section we show how HAMLET refines over-general and over-specific rules by changing their applicability conditions.

In the process planning domain plan length is usually not an accurate metric of plan quality, as different operators have different costs. Suppose the planner is given as a goal to drill two holes on different sides, 1 and 4, of a part and each hole requires a different diameter drill bit. In the initial state the part is being held ready to drill the hole on side 4, but the machine holds the drill bit suitable for the hole on side 1. One possible plan starts by releasing the part and holding it again in the orientation suitable for the hole in side 1, while keeping the tool in the drill spindle, and then drilling that hole. An alternative plan starts by switching the tool while keeping the part in place, and drilling first the hole in side 4. Both plans have the same length but may have different quality: if the tool can be switched automatically but holding the part requires human assistance, the first plan is costlier than the second. To obtain the better plan the goal of drilling the hole on side 4 has to be achieved before the goal of drilling the hole on side 1. Note that for a different plan evaluation function, namely one in which switching the tool is more expensive than holding the part again, the order to achieve the two goals should be the opposite. QUALITY is also able to learn goal preference rules that would produce the better plan in this case.

4.3 HAMLET: Incremental Inductive Learning of Control Knowledge

HAMLET views the planning episodes as an opportunity to learn control knowledge to improve both the planner's performance and the quality of the plans generated. Instead of relying on a comparison of pairs of complex plans as in QUALITY, HAMLET assumes that the learner is trained with simple problems for which the planner can explore the space of all possible plans. From this generated exhaustive search tree, HAMLET learns control rules that override the default choices of a particular search strategy and that direct the planner both to avoid planning failures and to generate plans of optimal quality. HAMLET learns incrementally correct control knowledge. It uses an initial deductive phase, where it generates a bounded explanation of the problem solving episode to create the control rules. This learned knowledge may be over-specific or over-general. Upon experiencing new problem solving episodes, HAMLET refines its control knowledge, incrementally acquiring increasingly correct control knowledge (Borrajo and Veloso, 1993; Borrajo and Veloso, 1994b). HAMLET evolves from previous work from one of the authors in NOLIMIT (Borrajo et al., 1992).

The inputs to HAMLET are a domain, a set of training problems, and a quality measure.[4] The output is a set of control rules. HAMLET has three main modules: the Bounded-Explanation module, the Inductive module, and the Refinement module.

The Bounded-Explanation module generates control rules from a PRODIGY search tree. These rules might be over-specific or over-general. The Induction module solves the problem of over-specificity by generalizing rules when analyzing positive examples. The Refinement module replaces over-general rules with more specific ones when it finds situations in which the learned rules lead to wrong decisions. HAMLET gradually learns and refines control rules converging to a concise set of correct control rules, i.e. rules that are individually neither over-general, nor over-specific.[5]

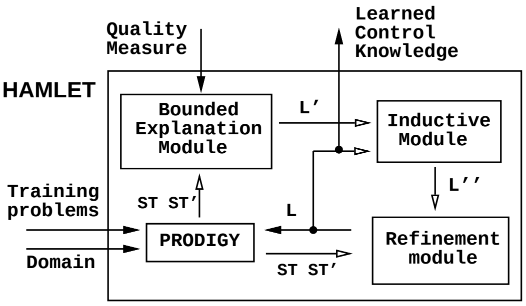
Figure 13(a) shows HAMLET's modules and their connection to PRODIGY. Here ST and ST$'$ are search trees generated by the PRODIGY planning algorithm, L is the set of control rules, L$'$ is the set of new control rules learned by the Bounded Explanation module, and L$''$ is the set of rules induced from L and L$'$ by the Inductive module.

Figure 13(b) outlines HAMLET's algorithm. If in a new training search tree, HAMLET finds new positive examples of the opportunities to learn control rules then it generates new rules and, when possible, induces more general rules. If negative examples are found of application of control rules, HAMLET refines them, i.e., generates more specific rules.

4.3.1 Bounded Explanation

The Bounded-Explanation module learns control rules by choosing important decisions made during the search for a solution and extracting the information that justifies these decisions from the search space. The explanation procedure consists of four phases: Labeling the decision tree, Credit assignment, Generation of control rules, and Parameterization.

Labeling the decision tree. HAMLET traverses the search tree generated by PRODIGY bottom-up, starting from the leaf nodes. It assigns three kinds of labels to each node of the tree: *success*, if the node corresponds to a correct solution plan; *failure*, if the node is a dead end in the search space; and *unknown*, if the planner did not expand the node, and thus we do not know whether this node can lead to a solution. Following the label of the leaf nodes, the algorithm propagates the labels up to the root of the search tree, using the the



(a)

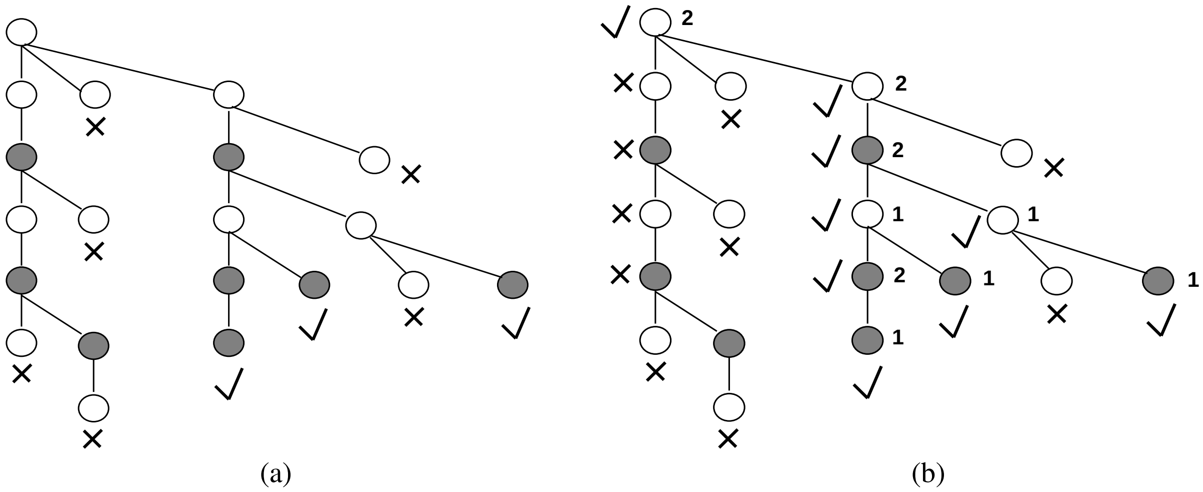
Let L refer to the set of learned control rules.
Let ST refer to a search tree.
Let P be a problem to be solved.
Let Q be a quality measure.
Initially L is empty.
For all P in the set of training problems
    ST = Result of solving P without any rules.
    ST' = Result of solving P with current set of rules L.
    If positive-examples-p(ST, ST′)
    Then L′ = Bounded-Explanation(ST, ST′,Q)
        L′′= Induce(L,L′)
    If negative-examples-p(ST, ST′)
    Then L=Refine(ST, ST′,L′′)
Return L

(b)

Figure 13: (a) shows HAMLET's high level architecture; and (b) shows a high-level description of HAMLET's learning algorithm

algorithm described in (Borrajo and Veloso, 1994a). The nodes in each solution path are also labeled with the length of the optimal solution that can be reached from this node.

Figure 14(a) shows an example of a typical search tree, in which each leaf node is labeled by the PRODIGY planner as success (√) or failure (X). Figure 14(b) shows how HAMLET propagates labels to the root of this tree. In general, there are several solutions to a problem. The grey nodes are the ones in which an operator was applied, i.e., an operator was moved to the head plan.



(a) (b)

Figure 14: (a) A typical PRODIGY search tree, corresponding to a search for more than one solution, where each leaf node is labeled as success (√) or failure (X); (b) The same tree after HAMLET labels it and attaches the optimal solution length out of each node.

Credit Assignment. The credit assignment is the process of selecting important branching decisions, for which learning will occur. It is done concurrently with labeling. HAMLET views a branching decision as a learning opportunity if this decision (1) leads to one of the best solutions and (2) differs from the default decision made by domain-independent heuristics. Figure 15 shows the learning opportunities that HAMLET would find in the search tree of Figure 14(b). The wide lines show the branching decisions that would be learned in the case of optimal learning.

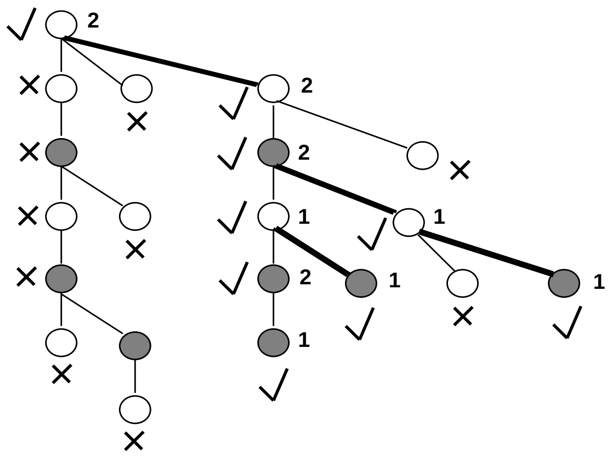


Figure 15: The learning opportunities in the tree of Figures 14(a) and (b).

Generation of control rules. At each decision point to be learned, HAMLET has access to information on the current state of the search and on the tail-plan information. This information is used by the generation module to create the applicability conditions, i.e. the *if*-parts of the control rules. The relevant features of the current state are selected using *goal regression*.

HAMLET learns four kinds of *select* control rules, corresponding to the PRODIGY's decisions. Each of the four kinds of control rules has a template for describing its *if*-parts. The templates have features that are instantiated with the particular state and tail-plan of the decisions to be learned. The complete set of the features can be found in (Borrajo and Veloso, 1994a). Since HAMLET learns several rules for each kind of control rule, the general representation of the learned knowledge is a disjunction of conjunctive rules.

Parameterization. After a rule is generated, HAMLET replaces specific constants inherited from the considered planning situation with variables of corresponding types. Distinct constants are replaced with differently named variables, and when the rule is applied, different variables must always be matched with distinct constants. The latter heuristic may be relaxed in the process of inductive generalization of the learned rules.

4.3.2 Inductive Generalization

The rules generated by the bounded explanation method may be over-specific as also noticed by (Etzioni and Minton, 1992). To address this problem, we use the Inductive Learning module, which generalizes the learned rules by analyzing new examples of situations where the rules are applicable. We have devised methods for generalizing over the following aspects of the learned knowledge: state; subgoaling dependencies; interacting goals; and type hierarchy (Borrajo and Veloso, 1994b).

In general, we can look at the inductive process within a target concept as it is shown in Figures 16. "WSP_i" stands for *Whole Set of Preconditions* for $rule_i$ and refers to the complete description of the current state and all available information about the tail-plan. Step (a) shows the Bounded-Explanation step in which the WSP_1 is generalized to R_1 by *goal regression* and parameterization. There is one WSP_i for each learning opportunity. Step (b) shows an inductive generalization from two bounded rules, R_1 and R_2. Step (c) shows another inductive generalization, from I_1 and R_3, generating a new rule I_2.

Since *goal regression* or the inductive operators can generate over-general rules, HAMLET keeps the learning history so that in can backtrack to generalization step, refining the learned knowledge as explained in the Section 4.3.4.

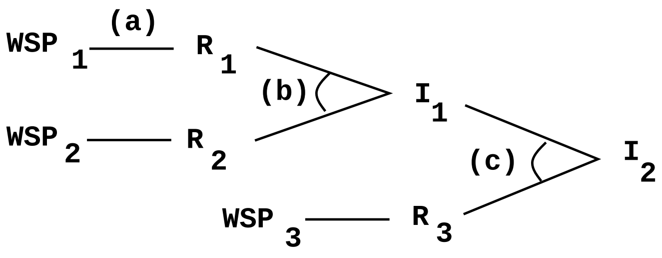


Figure 16: Three learning episodes of HAMLET within the same target concept. (a) shows the initial step of bounding the explanation from the *Whole Set of Preconditions* to a control rule R_1. (b) shows an inductive step from another rule R_2, generating induced rule I_1. (c) shows a second inductive step, generating I_2 from I_1 and R_3.

4.3.3 Example: Inducing New Control Rules

We will present now an example of how HAMLET induces from two rules a new one more general than the previous ones. The domain we use is a logistics-transportation domain (Veloso, 1992). In this domain, packages must be delivered to different locations in several different cities. Packages are carried within the same city in trucks and across cities in airplanes. At each city, there are several locations, such as post offices and airports. The domain consists of a set of operators to load and unload packages into and from the carriers at different locations, and to move the carriers between locations.

Consider the following problem solving situation illustrated in Figure 17. There are two cities, *city1* and *city2*, each one with one post-office and one airport. Initially, at *post-office1*, there are two packages, *package1* and *package3*, and two trucks, *truck1* and *truck3*. At *airport1* there is an airplane, *plane1*, and another package, *package2*. At *city2*, there is only one truck, *truck2*, at the city airport, *airport2*. There are two goals: *package1* must be at *post-office2*, and *plane1* at *airport2*. This problem is interesting because both the package and the airplane need to be moved to a different city. HAMLET will learn, among other things, that the package should be loaded into the airplane (or any other needed carrier) before the airplane moves.

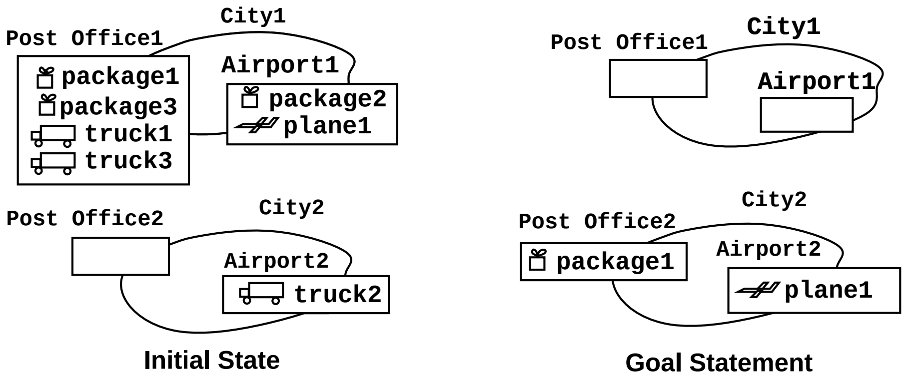


Figure 17: An illustrative example - initial state and goal statement.

The optimal solution to this problem is the sequence of steps in Table 8.

Notice that although in this example the optimal plan corresponds to this unique linearization, in general the learning procedure can reason about a partially-ordered dependency network of the plan steps.

HAMLET labels and assigns credit to decisions made in the search tree generated by PRODIGY. The rule in

(load-truck package1 truck1 post-office1),
(drive-truck truck1 post-office1 airport1),
(unload-truck package1 truck1 airport1),
(load-airplane package1 plane1 airport1),
(fly-airplane plane1 airport1 airport2),
(unload-airplane package1 plane1 airport2),
(load-truck package1 truck2 airport2),
(drive-truck truck2 airport2 post-office2),
(unload-truck package1 truck2 post-office2)

Table 8: Solution to the problem in Figure 17.

.

Table 9(a) is learned at one of the decisions made, namely when PRODIGY finds that it should delay moving the carriers until the package is loaded.[6] For HAMLET *target-goal* is an unachieved goal, *other-goals* is the rest of unachieved goals, and *prior-goal* is the goal that generated the current node of the search tree.

```
(control-rule select-inside-truck-1
 (if (and
    (target-goal (inside-truck <package1> <truck1>))
    (prior-goal (at-object <package1> <post-office2>))
    (true-in-state (at-truck <truck1> <post-office1>))
    (true-in-state (at-object <package1> <post-office1>))
    (true-in-state (same-city <airport1> <post-office1>))
    (other-goals ((at-truck <truck1> <airport1>)
                  (at-airplane <plane1> <airport2>)))))
 (then select goals (inside-truck <package1> <truck1>)))
```

(a)

```
(control-rule select-inside-truck-2
 (if (and
    (target-goal (inside-truck <package1> <truck1>))
    (prior-goal (at-object <package1> <airport2>))
    (true-in-state (at-truck <truck1> <airport1>))
    (true-in-state (at-object <package1> <post-office1>))
    (true-in-state (same-city <airport1> <post-office1>))
    (other-goals ((at-airplane <plane1> <airport2>)))))
 (then select goals (inside-truck <package1> <truck1>)))
```

(b)

Table 9: (a) Rule learned by HAMLET after applying the Bounded-Explanation (b) Rule learned by HAMLET from a second example.

The rule says that the planner should find a way of achieving *(inside-truck package1 truck1)* before moving the carriers, *plane1* and *truck1* (even though flying an empty airplane will achieve one of the two planning goals). This is a useful rule, since if PRODIGY decides to move a carrier before loading it, the carrier will have to be moved back to pick its load, thus resulting in a non-optimal plan. This rule is over-specific, since, among other things, the prior-goal does not have to be that the package be in a post-office, and *truck1* does not have to be in the same place as *package1* in order to work on one goal before another.

Suppose that PRODIGY solves a new problem in which *package1* and *plane1* have to go to *airport2*, and *truck1* is initially at *airport1*. HAMLET learns a new rule shown in Table 9(b). From the two rules in Table 9, the Induction module generates the rule presented in Table 10, which is the result of applying three inductive operators: *Relaxing the subgoaling dependency*, which removes the prior-goal from the *if*-part; *Refinement of goal dependencies*, which requires the other-goals to be the union of the other-goals of each rule; and *Finding common superclass*, which finds a common general type in the hierarchy of PRODIGY types, *location*, that is a common ancestor of the types *airport* and *post-office*. A complete list of inductive operators can be found in (Borrajo and Veloso, 1994a).

```
(control-rule select-inside-truck-3
  (if (and (target-goal (inside-truck <package1> <truck1>))
           (true-in-state (at-truck <truck1> <location1>))
           (true-in-state (at-object <package1> <post-office1>))
           (true-in-state (same-city <location1> <post-office1>))
           (other-goals ((at-truck <truck1> <location1>)
                         (at-airplane <plane1> <airport2>)))))
  (then select goals (inside-truck <package1> <truck1>)))
```

Table 10: Rule induced by HAMLET from the rules shown in Tables 9(a) and 9(b).

4.3.4 Refinement

HAMLET may also generate over-general rules, either by doing *goal regression* when generating the rules or by applying inductive operators. These over-general rules need to be refined. There are two main issues to be addressed: how to detect a negative example, and how to refine the learned knowledge according to it. A negative example for HAMLET is a situation in which a control rule was applied, and the resulting decision led to either a failure (instead of the expected success), or a solution worse than the best one for that decision.

When a negative example is found, HAMLET tries to recover from the over-generalization by refining the wrong rule. The goal is to find, for each over-general rule, a larger set of predicates that covers the positive examples, but not the negative examples. To speed up the convergence of the refinement, we have introduced informed elaborated ways of adding those features, by using an information content measure (Quinlan, 1983; Quinlan, 1990). We have adapted these methods to work incrementally. The refinement sorts the predicates that can be added to the *if*-part of the control rule according to their information potential. The information that a literal L_i supplies to a given target concept T_j can be computed by:

$$\textit{Information}(L_i, T_j) = \sum_{i=1}^{L} p(L_i, T_j) log\ p(L_i, T_j)$$

where L is the number of predicates in the domain, and $p(L_i, T_j)$ is the probability that a literal L_i belongs to a positive example of target concept T_j. To compute that measure, for each target concept, HAMLET keeps the frequency with which each literal has appeared in the state, or as a prior-goal, in the positive and negative examples of the target concept. A more detailed description of the refinement process can be found in (Borrajo and Veloso, 1994a).

As a general example, suppose that the rule I_2 in Figure 16 is found to be over-general, that is HAMLET found a negative example of its application. Figure 18 shows the process of refining it. In this case, one of its generating rules, I_1, was also over-general. The Refinement module backtracks, and refines I_1 with the rules that generated it, R_1, and R_2. R_1 was also over-general, so the HAMLET had to backtrack to consider also WSP_1, generating the refined rule RF_1. R_2 was not over-general, so it created a rule RF_2 from R_2 and I_1, then deleting I_1. Finally, R_3 was not over-general, so it generated a new rule RF_3, and deleted I_2. The dotted lines represent deleted links, the dotted boxes deleted rules, and the normal boxes the active rules after refinement.

4.3.5 Discussion

We have presented a learning system, HAMLET, that learns control rules that enables PRODIGY to efficiently obtain plans of good quality according to some user defined criteria. We believe that it is necessary to build

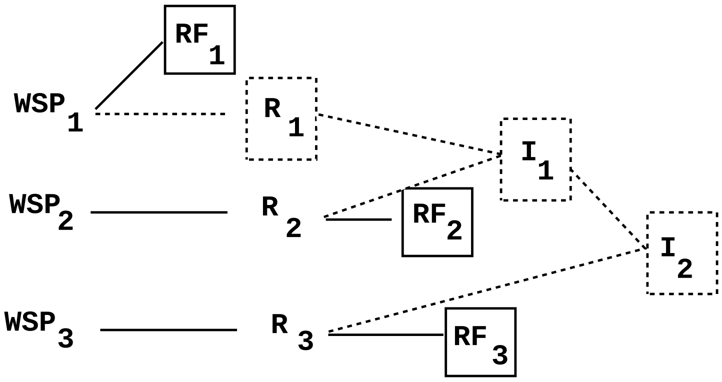


Figure 18: Graphical representation of the refinement process of an over-general control rule I_2.

this kind of strategy learning tools within existing problem solvers, since they should become more adaptive to changing environments.

HAMLET automatically analyzes the decision making search episode produced by the planner. It identifies the appropriate learning opportunities as the successful choices that should be repeated and the failed ones that should be avoided in future situations. Learning the correct generalized conditions responsible for the decisions from a single example is not tractable in real-world applications. Instead, HAMLET learns incrementally correct control knowledge. It uses an initial deductive phase, where it generates a bounded explanation of the problem solving episode to create the control rules. This learned knowledge may be over-specific or over-general. Upon experiencing new problem solving episodes, HAMLET refines its control knowledge, incrementally acquiring increasingly correct control knowledge.

4.4 Empirical Results

We are carrying out experiments to measure the impact of the learned knowledge on the quality of plans in several domains. In this section we present results in the process planning domain, for QUALITY, and in the logistics transportation, for HAMLET. Although we have not yet explored the characteristics of each method that make it suitable for a given domain, we show empirical results that allow PRODIGY overall to produce plans of good quality.

4.4.1 Plan Quality Improvement in the Process Planning Domain

We have implemented QUALITY and run preliminary experiments to evaluate the solution quality gained by the use of the learned control knowledge. Table 11 shows the effect of that knowledge on the solution cost, according to the evaluation function mentioned in Section 4.2.2, [7] over 70 randomly-generated problems in the process planning domain. Each column corresponds to a set of 10 problems with common parameters (number and type of goals, parts, etc) which determine the problem difficulty and the usefulness of quality-enhancing control knowledge. In many of the training and test problems the planner did not require control knowledge to obtain a good solution and therefore for each problem set we have only recorded the problems where the solution was actually improved. The third row corresponds to the cost of the solutions obtained by the planner when it lacks any quality-enhancing control knowledge. The fourth row shows the cost of the solutions obtained using the learned control rules. The rules were learned from a different set of 60 randomly generated problems with the same parameters than for sets 1 to 6 of Table 11. In only 41 of these problems the solution could be improved and therefore learning was invoked. The smaller quality improvement obtained for set 7 is due to the fact that the training phase did not include problems with

Problem set (10 probs per set)	1	2	3	4	5	6	7
# problems with improvement	3	9	3	10	10	4	9
Without learned control knowledge	107	202	190	431	362	442	732
With learned control knowledge	91	132	166	350	220	409	665
Cost decrease	44%	48%	33%	24%	47%	17%	8%

Table 11: Improvement on the quality of the plans obtained for 70 randomly-generated problems in the process planning domain.

similar parameters and therefore some of the control knowledge relevant to those problems had not yet been learned. The number of nodes was also reduced by using the learned control knowledge, due to shorter solution lengths. The total CPU time was also reduced, but still we plan to further analyze the cost of using learned knowledge, and the possible tradeoff between the matching cost and the savings obtained by using the learned knowledge instead of doing a more exhaustive search until a reasonably good solution is found according to the evaluation function. We are exploring the effect of this learning mechanism on other domains and on other types of evaluation functions.

4.4.2 Plan Quality Improvement in the Logistics Domain

We trained HAMLET with 400 simple randomly generated problems of one and two goals, and up to three cities and five packages, using the solution length as the quality measure. HAMLET generated 17 control rules. Then, we randomly generated 500 testing problems of increasing complexity. Table 12 shows the results of those tests. We varied the number of goals in the problems from 1 up to 50, and the maximum number of packages from 5 up to 50. Notice that problems with 20 and 50 goals pose very complex problems to find optimal or good solutions.

Test sets		Unsolved problems		Solved problems					
				Better solutions		Solution length		Nodes explored	
Number of Goals	Number of Problems	without rules	with rules	without rules	with rules	without rules	with rules	without rules	with rules
1	100	5	1	0	6	322	309	2139	1649
2	100	15	10	0	16	528	494	3485	2567
5	100	44	29	0	24	878	826	5275	4313
10	100	68	38	0	19	865	792	3932	3597
20	75	62	41	0	7	505	475	2229	2039
50	25	24	16	0	0	34	34	144	141
Totals	500	218	135	0	72	3132	2930	17204	14306

Table 12: Table that shows the results on increasingly complex problems in the logistics domain.

The results show a remarkable decrease in the number of unsolved problems using the learned rules, as well as a great number of problems in which the solution generated using the rules is of better quality, accounting only the problems solved by both configurations. The time bound given was 150*(1+mod(number-goals,10)) seconds. The running times also decreased using the rules, but not significantly. We are currently developing efficient methods for organizing and matching the learned control rules. We consider this organization essential to the overall learning process to avoid a potential utility problem due to inefficient matching (Doorenbos and Veloso, 1993).

4.5 Related Work

Most of the work on plan quality has been on the relationship between plan quality and goal interactions. (Wilensky, 1983) explores this relationship in detail. Several planners for the process planning domain deal with the quality of plans (Hayes, 1990; Nau and Chang, 1985) using domain-dependent heuristics. Other work focuses on post-facto plan modification (Karinthi et al., 1992; Foulser et al., 1992) by merging plans for individual subgoals taking advantage of helpful goal interactions. PRODIGY uses instead control knowledge at problem solving time and is able to learn from experience in a domain-independent fashion. In (Ruby and Kibler, 1992) the learned control knowledge is acquired by an underlying case-based learner. (Iwamoto, 1994) has developed an extension to PRODIGY to solve optimization problems and an EBL method to learn control rules to find near-optimal solutions in LSI design. The quality goals are represented explicitly and based on the quality of the final state instead of that of the plan. This is equivalent to our use of a quality evaluation function. The learning method is similar to QUALITY in that it compares two solutions of different quality, builds an explanation, and learns operator preferences. QUALITY however makes use of the quality evaluation function to build the explanation, and learns in addition bindings and goal preference rules. As in our case, the learned knowledge may be overgeneral and is refined upon further experience. The method does not allow user guidance as it uses exhaustive search until the quality goal is satisfied to find the best solution. This is possible because of the relatively small size of the search space of the examples used.

Some EBL for problem solving systems (Golding et al., 1987) learn from externally provided guidance. When used, the guidance takes the form of a plan reducing the need for the user to understand the problem solving process. However our method does not rely on such guidance, as since the quality evaluation function is available, the system continues searching for better solutions and then learns about which paths are better than others. R1-Soar (Rosenbloom et al., 1985) uses a similar approach for optimizing a computer configuration task. Chunking is used indistinctly to reduce the search and to prefer better solutions. QUALITY explains the difference in cost between the two paths rather than simply the union of what mattered along the two paths and therefore may be able to build more general rules.

There have been a number of systems that learn control knowledge for planning systems, most of them are oriented towards improving search efficiency and not plan quality. Most of these speedup learning systems have been applied to problem solvers with the linearity assumption, such as the ones applied to Prolog or logic programming problem solvers (Quinlan, 1990; Zelle and Mooney, 1993), special-purpose problem solvers (Mitchell et al., 1983; Langley, 1983), or other general-purpose linear problem solvers (Etzioni, 1990; Fikes and Nilsson, 1971; Leckie and Zukerman, 1991; Minton, 1988b; Pérez and Etzioni, 1992). These problem solvers are known to be incomplete and unable of finding optimal solutions (Rich, 1983; Veloso, 1989).

Learning strategy knowledge is a necessary step towards the application of today's computer's technology to real world tasks. Most of the current strategy learning systems eagerly try to explain correctly the problem solving choices from a single episode or from a static analysis of the domain definition. Therefore they rely heavily on a complete axiomatic domain theory to provide the support for the generalization of the explanation of one episode to other future situations. This requirement and this explanation effort is not generally acceptable in real-world applications that are incompletely specified and subject to dynamic changes.

If we remove the linearity assumption, we are dealing with nonlinear problem solvers. This kind of problem solvers are needed to address real world complex problems. Some nonlinear planners search the plan space by using partially-ordered plans (Chapman, 1987; McAllester and Rosenblitt, 1991). Others remove the linearity assumption by fully interleaving goals, searching in the state space, and keeping totally-ordered plans (Veloso, 1989; Warren, 1974). Most of the approaches do backward chaining, although there are some approaches that use forward chaining, (Bhatnagar, 1992; Laird et al., 1986). In general, there have not been as many learning systems applied to nonlinear problem solving. Some approaches are (Bhatnagar, 1992; Kambhampati and Kedar, 1991; Laird et al., 1986; Ruby and Kibler, 1992; Veloso, 1992). Our work addresses learning in the context of nonlinear planning.

A similar lazy approach, is followed by LEBL (Tadepalli, 1989) (Lazy Explanation Based Learning). The main difference is that while LEBL refines the knowledge introducing exceptions, QUALITY establishes priorities, and HAMLET modifies the control rules themselves adding or removing their applicability conditions. LEBL applies to games, while we use it for general task planning, and LEBL does not consider plan quality.

5 PRODIGY – Discussion and Current Characteristics

We have been doing research in the PRODIGY architecture for several years now. The main characteristic of the PRODIGY research project has been that of addressing planning as a *glass box* decision-making process to be integrated with learning. In essence, in PRODIGY we view all planning decisions as learning opportunities, some of which we have addressed in depth and others that are part of our ongoing research agenda:

- Where to expand the plan under construction
- Whether to commit to an operator application or continue subgoaling
- Which goal(s) or subgoal(s) to work on next
- What operators to select in order to reduce the selected goal(s)
- What bindings in the state to select in operator instantiation
- Where to direct backtracking in the search space upon failure
- Whether or not to retain a new control-rule or a new planning case
- Whether to plan using the domain theory or by replaying retrieved cases
- Whether to experiment or forge ahead when there is evidence of an incompletely specified operator
- Whether to interact with a human expert user or run in fully-automated mode.

Our long-term goal has always been to enable PRODIGY to address increasingly-complex real-world planning problems. Towards this goal, we have been progressively enhancing the planner and the learning algorithms to cope with the challenges presented by the real world.

Table 13 shows the evolution of the PRODIGY system from its first problem solving version in PRODIGY1.0 to the current planner and learning system, PRODIGY4.0. The table enumerates some of PRODIGY's characteristics along the dimensions of the domain representation, the search strategies and the learning methods.

This table captures most of PRODIGY's approaches and solutions to incrementally more challenging planning tasks and problems. Next we discuss a few of the issues underlying some of these evolutionary processes.

5.1 Representation – Domain and Control Knowledge

One of the contributions of Steven Minton's dissertation in terms of the initial design of the PRODIGY system solver was to clearly distinguish between domain and control knowledge. Far from being obvious, such a distinction has been repeatedly overlooked in the planning literature. Yet, it is precisely this clean and consistent distinction that has enabled the PRODIGY group to develop and test a rich variety of learning mechanisms, starting with speedup learning to maximize planning-efficiency, through plan-quality learning to maximize plan-execution efficiency, and into factual-knowledge refinement of incomplete or incorrect planning operators, and most recently into entirely new-operator learning.

		Prodigy1.0	Prodigy2.0	NoLimit (Prodigy3.0)	Prodigy4.0
Planning Domain and Control Knowledge	Objects organized in a class hierarchy			√	√
	Conjunctive preconditions	√	√	√	√
	Disjunction, negation, existential and universal quantification in preconditions		√	√	√
	Functions constraining bindings		√	√	√
	Regular effects: add, delete lists	√	√	√	√
	Conditional effects		√	√	√
	Inference rules		√	√	√
	Control rules		√	√	√
	Control guidance from planning episodes			√	√(prep)
Planning Search Strategies	State-space search; means-ends analyis; backward chaining	√	√	√	√
	Linear: goal independence assumption	√	√		
	Nonlinear: full interleaving of goals and subgoals			√	√
	Chronological backtracking	√	√	√	√
	Dependency-directed backtracking in some situations			√	√
	Large set of domain-independent search heuristics				√
	Planning with abstraction levels		√		√
	Anytime planning				√
	Incremental matching for operators		√		
	Rete matcher for control rules			√	
	Rete-based matcher for operators and control rules				√
Learning	Control Knowledge		EBL Alpine Static Dynamic	Analogy	Hamlet Analogy (prep)
	Domain knowledge		Apprentice Experiment	Apprentice	Observe
	Plan quality				Quality Hamlet

Table 13: Evolution of PRODIGY – towards planning and learning in real-world planning problems.

The primary distinction between previous AI systems and the PRODIGY system that also make a domain-control distinction is that the former typically employ an impoverished "second-class" representation status for the control knowledge. In particular, game-playing systems express their control knowledge as a numeric evaluation function, usually a linear weighted sum. Control knowledge in macro-operators consists of nothing more than a fancy concatenation operation.

PRODIGY utilizes a typed-FOL representation for its domain knowledge (operators and inference-rules), and an identical representation augmented with meta-predicates is used to represent control knowledge in the form of meta-rules or cases that guide decision making at the decision points listed above. Although the languages in which control and domain knowledge are expressed are essentially the same, the role each plays internal to the system is markedly different. Domain knowledge lays out the legal moves and control knowledge strategies on which legal moves to apply in order to best achieve the given goals. The fact that arbitrary reasoning can occur at the control level, and can apply to all decision points is a fundamental tenet of the PRODIGY architecture, and one that has served us well.

The advantages of the factual-control knowledge distinction are:

- *Modularity* of the domain and control representation languages, making it easy to add or modify new knowledge of each kind in system development.
- *Reification* of the control knowledge as explicit data structures, rather than buried in the internals of the interpreter permitting direct acquisition and clear glass-box examination of control knowledge.
- *Selectivity* in building learning modules. Each learning module can focus specifically on certain types of knowledge (e.g. planning efficiency) as applied to certain types of decision points.
- *Compositionality* of the acquired control knowledge. In theory, multiple learning methods can co-exist in any given task domain and contribute synergisticly to improved performance as experience accumulates. This theory, so far, has been reduced to practice only in a pair-wise manner (at most two learning systems coexisting synergisticly). But, larger-scale multi-module learning is in our longer-term research agenda.

5.2 Multiple Planning Procedures

The modularity of PRODIGY's structure allows for a large degree of versatility. As a result, PRODIGY4.0 emulates several different planning search strategies, by controlling its domain-independent search parameters. The planner can employ breadth-first, depth-first, or best-first search, as well as a controlled state-space search and means-ends analysis. Moreover, multi-level hierarchical planning and plan-refinement, as well as case-based tracking are supported. PRODIGY did not always have these different search mechanisms. Currently, PRODIGY4.0 can also emulate linear planning, and nonlinear planning by full interleaving of goals and subgoals permitting controlled experimentation. Early versions of PRODIGY were not as flexible. However, as we increased the variety and complexity of the domains that PRODIGY would apply to, we noticed that different planning strategies were better suited for different classes of domains. Additionally, we strove for comparisons with other planners, and hence permitting PRODIGY to plan in multiple modes became quite useful (Stone et al., 1994; Veloso and Blythe, 1994).

5.3 Domain Generality

PRODIGY has been applied to a wide range of planning and problem-solving tasks: robotic path planning, the blocksworld, several versions of the STRIPS domain, matrix algebra manipulation, discrete machine-shop planning and scheduling, computer configuration, logistics transportation planning, and several others.

The PRODIGY architecture is built as a domain-independent AI system. If the user is able to describe the domain in a series of operators in PRODIGY's domain representation language, then the PRODIGY planner will try to find a solution to problems in that domain. It is an empirical question to identify the characteristics of the domains that can comply to PRODIGY's assumptions.

Even if we subscribe to this philosophy that there should be an underlying domain-independent substrate, we are aware through our extensive empirical experiments of the domain-dependence of PRODIGY's multiple domain-independent search strategies. We believe that there is no single search heuristic that performs more efficiently than others for all problems or in all domains (Stone et al., 1994). One of our current research interests is on learning clusters of domains that map appropriately to the different PRODIGY's default search strategies, while still permitting the learning mechanisms to improve performance in the given domain, to handle exceptions, and to cache reasoning paths.

5.4 Scalability

Any integrated architecture must address increasingly large tasks, whether its objective is to model human cognition or to build useful knowledge-based systems for complex tasks. Scalability can be calibrated in multiple ways, but all relate to efficient behavior with increasing complexity. In PRODIGY we seek to achieve a reasonable measure of scalability in these dimensions:

- Size of the domain: total number of objects, attributes, relations, operators, inference rules, etc.
- Size of the problem: number of steps in the solution plan, number of conjuncts in the goal expression, size of the visited search space, etc.
- Variety: number of qualitatively different actions and object types in the domain.
- Perplexity: average fan-out at every decision point in the search space (with and without learned control knowledge).
- Stability: whether the domain involved only deterministic actions or involves unpredictable or exogenous events.

The learning techniques strive to reduce the visited search space in future problems with respect to the virtual (complete) search space.

5.5 Cognitive Fidelity

As mentioned earlier, the PRODIGY project strives to produce a useful, scalable, and maintainable reasoning and learning architecture. Where this matches human cognition, it is so by accident, by the limited imagination of the PRODIGY designers, or perhaps because the human mind has indeed optimized such aspects of cognition. In all other aspects, the goal is to produce the most effective system for problem solving, planning and learning, and the choice of cognitive approach is secondary to that goal. Here we enumerate a few additional ways in which PRODIGY differs from human thought and from other cognitive architectures:

- PRODIGY deliberates on any and all decisions: which goal to work on next, which operator to apply, what objects to apply the operator to, where to backtrack given local failure, whether to remember newly acquired knowledge, whether to refine an operator that makes inaccurate predictions, and so on. It can introspect fully into its decision cycle and thus modify it at will. This is not consistent with the human mind, yet it is an extremely useful faculty for rapid learning.
- We do not know whether the human mind compartmentalizes distinct cognitive abilities, or distinct methods of achieving similar ends (such as our multiple learning methods). However, modularization at this level is a sound engineering principle, and therefore we have adhered to it. Integration is brought about by sharing both a uniform knowledge representation, and a common problem-solving and planning engine. Other systems, such as SOAR, take on a monolithic structure. There is only one learning mechanism, chunking, which can never be turned off or even modulated. Which is better? Clearly, we believe the former to be superior – but only from engineering principles rather than psychological ones.

5.6 Implementation Lessons Learned

In PRODIGY we start addressing each complicated phenomenon by addressing first simpler versions. When we extend the simple versions to address the more complicated phenomena, we occasionally find the need to reimplement the system in part or in whole. The process becomes the sequence of simplification, research and development of approximate learning and planning methods, success for the simplified task domains, extension, augmentations and patching, and, when necessary, reimplementation.

We believe that this process is inevitable in building large AI systems. A lesson we learned however along this process is that we should always keep the full-fledged phenomena in mind and keep the system as flexible as possible. In many situations it has not been easy to transfer the capabilities of one version of the system to the new system that exhibits brand new capabilities in orthogonal dimensions. Fortunately, our latest system, PRODIGY4.0, has proven more flexible than earlier implementations, and has handled a far wider variety of phenomena. So, perhaps we have learned the lesson, even if there are no guarantees, that a yet newer, more powerful system implementation will be required to address new more challenging phenomena.

5.7 Regrets, Lessons, and Afterthoughts

Not every decision in PRODIGY has proven successful, some even appear ill-conceived in the harsher light of informed hindsight. Most of the ill-fated decisions, fortunately, were of an implementational nature, and were correctable given PRODIGY's modular architecture. Some, however, were of a more conceptual nature, and it is these that we enumerate here:

- *The Hidden Control Knowledge Dilemma* – Early on in PRODIGY we designed operator preconditions with a lexically-fixed expansion order. That is, operator preconditions were subgoaled strictly left-to-right. Far from a mere-convenience, this simple design decision created a control point (subgoal ordering) not open to the learning mechanisms, and hence made performance very dependent on correct initial encoding. Taking out this "feature" and others like it that correspond to implicit control knowledge introduced performance degradation in the short run, but enhanced learnability of control knowledge in the long run. In essence, we learned not to deviate from the glass-box philosophy that all decisions are deliberate, explicit, and all comprise opportunities to learn.

- *The Nonlinear EBL Myth* – The initial explanation-based speedup learning techniques in PRODIGY, namely PRODIGY/EBL, STATIC and DYNAMIC, were developed in the context of the linear PRODIGY2.0. The methods required a complete axiomatization of each planning domain to support the generalization of the explanations of successes and failures of a single particular search episode. The explanation-based algorithms eagerly try to explain correctly the problem solving choices from a single episode or from a static analysis of the domain definition. Therefore they rely heavily on the complete axiomatic domain theory to provide the support for the generalization of the explanation of one episode to other future situations.

  In the process of trying to apply these techniques directly as designed to the nonlinear PRODIGY4.0, we accumulated interesting examples where the methods failed in learning correct or useful control knowledge. In fact, in nonlinear problem solving, as developed in PRODIGY, it is not feasible to obtain a perfect explanation of why some decisions are correct. The explanation-based methods rely heavily on an axiomatization of the problem solver and planning domain, which are not easily acquired from a user or from automatic domain analysis. We therefore developed alternative approaches that learn control knowledge lazily in the form of planning cases that are replayed by analogy, in PRODIGY/ANALOGY, and by inductively and incrementally refining learned knowledge as in HAMLET.

- *The Ubiquitous Version Control Problem* – Some learning mechanisms depended upon the internal data structures of the planner, making their application to a newer more powerful version of the system problematical. We painfully learned the software engineering lessons of modularity, localization of data paths in access functions, and careful version control of all the above. Unfortunately this lesson is not being as universally applied in all parts of our project as we would like.

- *The Simpler-Problem Generation Detail* – We discovered early on in the creation of learning by derivational analogy that it is often far more efficient to solve simple versions of problems and then expand their solution into the full-fledged versions of the problems – rather than attacking the latter at the start. Naively, we decided to build an add-on module for problem simplification. That was several years ago, and the module still does not exist. In fact, we stumbled into an open research problem, recognized by George Polya (Polya, 1945), of characterizing in general what are useful simplifications of arbitrary problems. We have made headway, but are still exploring.

In conclusion, we have found the PRODIGY system to be an extremely useful research vehicle for planning, learning and scaling. In spite of some flaws, it has permitted nearly a dozen graduate students to explore questions that would have required far longer than a PhD period, had there not been the PRODIGY platform to use as a launching pad, or a booster. Moreover, it has permitted comparisons and analyzes of multiple learning techniques on problems, common encodings, and with common planning substrates. The research continues, with a current greater emphasis on the planning methods, learning the map from problem and domain to planning search strategy, and on learning at the knowledge level.

6 Conclusion

PRODIGY's architecture integrates planning and different learning mechanisms in a robust reasoning framework. In this article we introduced the PRODIGY system. We presented PRODIGY's planner and overviewed some of the early learning modules. Learning in PRODIGY addresses three different aspects of performance improvement of an AI reasoning system: learning at the control level, in which we learn to improve the efficiency of the planner's search; learning at the domain level, in which the domain theory, i.e., the set of planning operators, is acquired and/or refined when and if incompletely specified; and learning plan quality, in which we focus on acquiring planning knowledge that guides the better to generate plans of good quality according to some previously specified quality metric. We presented in particular the recently developed learning strategies to acquire control knowledge to guide the planner to produce solutions of good quality.

Finally we discuss the overall characteristics of the PRODIGY system and present their evolution along the several years of research towards producing a large and robust planning and learning AI system. PRODIGY architecture is a demonstration, and we also show in this article empirical evidence, that the integration of planning and learning greatly increases the solvability horizon of a planning system. Learning allows the interpretation of planning experience in solving simple problems to be transferred to problems of considerable complexity that a planner may not be able to address directly. PRODIGY has been applied to complex planning domains. Our research goal in PRODIGY is to use the powerful integration of planning and learning to handle problems increasingly close to real-world problems.

References

Bhatnagar, N. (1992). Learning by incomplete explanations of failures in recursive domains. In *Proceedings of the Machine Learning Conference 1992*, pages 30–36.

Blythe, J. and Veloso, M. M. (1992). An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, MD.

Borrajo, D., Caraça-Valente, J. P., and Pazos, J. (1992). A knowledge compilation model for learning heuristics. In *Proceedings of the Workshop on Knowledge Compilation of the 9th International Conference on Machine Learning*, Scotland.

Borrajo, D. and Veloso, M. (1993). Bounded explanation and inductive refinement for acquiring control knowledge. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 21–27, Amherst, MA.

Borrajo, D. and Veloso, M. (1994a). HAMLET: Incremental deductive and inductive learning of local control knowledge for nonlinear problem solving. Technical Report CMU-CS-94-111, School of Computer Science, Carnegie Mellon University.

Borrajo, D. and Veloso, M. (1994b). Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning, ECML-94*, pages 64–82. Springer Verlag.

Carbonell, J. G., Blythe, J., Etzioni, O., Gil, Y., Joseph, R., Kahn, D., Knoblock, C., Minton, S., Pérez, A., Reilly, S., Veloso, M., and Wang, X. (1992). PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Carnegie Mellon University.

Carbonell, J. G. and Gil, Y. (1990). Learning by experimentation: The operator refinement method. In Michalski, R. S. and Kodratoff, Y., editors, *Machine Learning: An Artificial Intelligence Approach, Volume III*, pages 191–213. Morgan Kaufmann, Palo Alto, CA.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378.

Descotte, Y. and Latombe, J.-C. (1985). Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27:183–217.

Doorenbos, R. B. and Veloso, M. M. (1993). Knowledge organization and the utility problem. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 28–34, Amherst, MA.

Doyle, L. E. (1969). *Manufacturing Processes and Materials for Engineers*. Prentice-Hall, Englewood Cliffs, NJ, second edition.

Etzioni, O. (1990). *A Structural Theory of Explanation-Based Learning*. PhD thesis, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-90-185.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–301.

Etzioni, O. and Minton, S. (1992). Why EBL produces overly-specific knowledge: A critique of the prodigy approaches. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 137–143.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Fink, E. and Veloso, M. (1994). PRODIGY planning algorithm. Technical Report CMU-CS-94-123, School of Computer Science, Carnegie Mellon University.

Foulser, D. E., Li, M., and Yang, Q. (1992). Theory and algorithms for plan merging. *Artificial Intelligence*, 57:143–181.

Gil, Y. (1991). A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Gil, Y. (1992). *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-92-175.

Gil, Y. and Pérez, M. A. (1994). Applying a general-purpose planning and learning architecture to process planning. In *Working notes of the AAAI 1994 Fall Symposium Series, Symposium on Planning and Learning: On to Real Applications*.

Golding, A., Rosenbloom, P. S., and Laird, J. E. (1987). Learning general search control from outside guidance. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, pages 334–337, Milan, Italy.

Hayes, C. (1990). *Machining Planning: A Model of an Expert Level Planning Process*. PhD thesis, The Robotics Institute, Carnegie Mellon University.

Iwamoto, M. (1994). A planner with quality goal and its speedup learning for optimization problem. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 281–286, Chicago, IL.

Joseph, R. (1989). Graphical knowledge acquisition. In *Proceedings of the 4th Knowledge Acquisition For Knowledge-Based Systems Workshop*, Banff, Canada.

Joseph, R. L. (1992). *Graphical Knowledge Acquisition for Visually-Oriented Planning Domains*. PhD thesis, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-92-188.

Kambhampati, S. and Kedar, S. (1991). Explanation based generalization of partially ordered plans. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 679–685.

Karinthi, R., Nau, D. S., and Yang, Q. (1992). Handling feature interactions in process planning. *Applied Artificial Intelligence*, 6(4):389–415. Special issue on AI for manufacturing.

Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, 68.

Knoblock, C. A., Minton, S., and Etzioni, O. (1991). Integrating abstraction and explanation based learning in PRODIGY. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 541–546.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46.

Langley, P. (1983). Learning effective search heuristics. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 419–421.

Leckie, C. and Zukerman, I. (1991). Learning search control rules for planning: An inductive approach. In *Proceedings of Machine Learning Workshop*, pages 422–426.

McAllester, D. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639.

Minton, S. (1988a). *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston, MA.

Minton, S. (1988b). *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. PhD thesis, Computer Science Department, Carnegie Mellon University. Available as technical report CMU-CS-88-133.

Minton, S., Knoblock, C. A., Kuokka, D. R., Gil, Y., Joseph, R. L., and Carbonell, J. G. (1989). PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University.

Mitchell, T. M., Utgoff, P. E., and Banerji, R. B. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning, An Artificial Intelligence Approach*, pages 163–190. Tioga Press, Palo Alto, CA.

Nau, D. S. and Chang, T.-C. (1985). Hierarchical representation of problem-solving knowledge in a frame-based process planning system. Technical Report TR-1592, Computer Science Department, University of Maryland.

Peot, M. A. and Smith, D. E. (1993). Threat-removal strategies for partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 492–499, Washington, DC.

Pérez, M. A. and Carbonell, J. G. (1993). Automated acquisition of control knowledge to improve the quality of plans. Technical Report CMU-CS-93-142, School of Computer Science, Carnegie Mellon University.

Pérez, M. A. and Etzioni, O. (1992). DYNAMIC: A new role for training problems in EBL. In Sleeman, D. and Edwards, P., editors, *Proceedings of the Ninth International Conference on Machine Learning*, pages 367–372. Morgan Kaufmann, San Mateo, CA.

Pérez, M. A. and Veloso, M. M. (1993). Goal interactions and plan quality. In *Preprints of the AAAI 1993 Spring Symposium Series, Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, pages 117–121, Stanford University, CA.

Polya, G. (1945). *How to Solve It*. Princeton University Press, Princeton, NJ.

Quinlan, J. R. (1983). Learning efficient classifiation procedures and their application to chess end games. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning, An Artificial Intelligence Approach, Volume I*. Morgan Kaufman.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5:239–266.

Rich, E. (1983). *Artificial Intelligence*. McGraw-Hill, Inc.

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. (1985). *R1-Soar*: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5):561–569.

Ruby, D. and Kibler, D. (1992). Learning episodes for optimization. In *Proceedings of the Machine Learning Conference 1992*, pages 379–384, San Mateo, CA. Morgan Kaufmann.

Stone, P., Veloso, M., and Blythe, J. (1994). The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 694–700, San Mateo, CA. Morgan Kaufmann.

Tate, A. (1977). Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–900, Cambridge, MA.

Veloso, M. and Blythe, J. (1994). Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 170–175.

Veloso, M. M. (1989). Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University.

Veloso, M. M. (1992). *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-92-174. A revised version of this manuscript will be published by Springer Verlag, 1994.

Veloso, M. M. and Carbonell, J. G. (1990). Integrating analogy into a general problem-solving architecture. In Zemankova, M. and Ras, Z., editors, *Intelligent Systems*, pages 29–51. Ellis Horwood, Chichester, England.

Veloso, M. M. and Carbonell, J. G. (1993). Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278.

Wang, X. (1992). Constraint-based efficient matching in PRODIGY. Technical Report CMU-CS-92-128, School of Computer Science, Carnegie Mellon University.

Wang, X. (1994). Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, pages 335–340, Chicago, IL.

Warren, D. (1974). WARPLAN: A system for generating plans. Technical report, Department of Computational Logic, University of Edinburgh. Memo No. 76.

Wilensky, R. (1983). *Planning and Understanding*. Addison-Wesley, Reading, MA.

Zelle, J. and Mooney, R. (1993). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*.