Formalizing the PRODIGY Planning Algorithm

Eugene Fink *
eugene@cs.cmu.edu
http://www.cs.cmu.edu/~eugene

Manuela Veloso *
veloso@cs.cmu.edu
http://www.cs.cmu.edu/~mmv

Computer Science Department, Carnegie Mellon University
Pittsburgh, Pennsylvania 15213, USA

Abstract. The PRODIGY project is primarily concerned with the integration of planning and learning. Members of the PRODIGY research group have developed many learning algorithms for improving planning efficiency and plan quality, and for automatically acquiring knowledge about the properties of planning domains. The details of the PRODIGY planning algorithm, however, have not been described in the literature.

We present a formal description of the planning algorithm used in the current version of the PRODIGY system. The algorithm is based on an interesting combination of backward-chaining planning with simulation of plan execution. The backward-chainer selects goal-relevant operators and then the planner simulates the application of these operators to the current state of the world. The system can use different backward-chaining algorithms, two of which are presented in the paper.

1 Introduction

PRODIGY is an integrated planning and learning system, which includes a planning algorithm and procedures for learning control knowledge to improve planning efficiency and plan quality [Veloso *et al.*, 1995]. PRODIGY is able to learn control rules [Minton, 1988, Veloso and Borrajo, 1994], generate abstraction hierarchies [Knoblock, 1994], use analogical reasoning to recognize and exploit similarities between planning problems [Veloso, 1994], and conduct experiments to acquire missing knowledge about the properties of a planning domain [Gil, 1991a, Wang, 1994].

A solution to a planning problem is represented in the PRODIGY system as a sequence of operators, which can be applied to the initial state of the world to achieve some state that satisfies the goal statement. PRODIGY's core, the planning algorithm, is designed to search for such sequences of operators.

The PRODIGY system includes a variety of mechanisms that enhance the operator-based planner. The system is able to guide the planner's search using control rules, hierarchical organization of operators, and a library of planning cases. The knowledge used in guiding the

*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.

search may be encoded by the user or learned automatically. The development of methods for learning the guiding knowledge automatically is the main purpose of the PRODIGY project.

The PRODIGY planning algorithm has been improved over years. The initial algorithm, PRODIGY2.0 [Minton *et al.*, 1989], was succeeded by NOLIMIT [Veloso, 1989] and then by the current PRODIGY4.0 [Carbonell *et al.*, 1992, Veloso *et al.*, 1995]. All versions of PRODIGY were developed by members of the PRODIGY research group at Carnegie Mellon University. We list the authors of the system in the acknowledgements section.

PRODIGY4.0 is a nonlinear planning system, which uses means-ends analysis to reason about multiple goals and multiple alternative plans achieving the goals. The dynamic goal selection enables the planner to interleave plans and exploit common subgoals.

The planning system is based on a combination of backward-chaining with simulation of plan execution, similar to forward-chaining planning. The system consists of two parts: a simulator of the plan execution and a backward-chaining algorithm. The system may use different backward-chaining algorithms, two of which are described in the paper.

The backward-chaining algorithm is responsible for goal-directed reasoning, whereas the execution-simulator enhances the goal-directed search with elements of forward-chaining. The execution-simulator searches among states of the world that can be achieved from the initial state by applying different operators. Unlike usual forward-chainers, the simulator uses only goal-relevant operators, selected by the backward-chaining algorithm.

The goal of our paper is to give a formal description of the PRODIGY planner. We describe the backward-chaining and execution-simulating planning algorithms used in PRODIGY and show how this algorithms are combined into a single system.

2 Domain and Problem Definitions

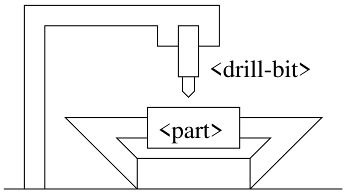
We define a *planning domain* by a set of object classes and a library of operators that act on objects of these classes. PRODIGY's language for describing operators is based on the STRIPS domain language [Fikes and Nilsson, 1971], extended to express disjunctive preconditions, universal and existential quantification, functions, and conditional effects [Carbonell *et al.*, 1992].

An operator is defined by its *preconditions* and *effects*. The description of preconditions and effects of an operator can specify classes of variables and restrict the values of variables by logical formulas. The *preconditions* of an operator are represented as a logical expression containing conjunctions, disjunctions, negations, and universal and existential quantifiers. The *effects* of an operator are a list of predicates to be added to or deleted from the current state of the domain.

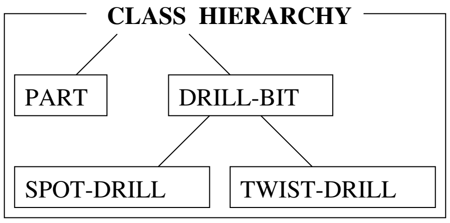
In Figure 1, we show an example of a planning domain, a simplified version of the drilling operations in the PRODIGY Process-Planning Domain [Gil, 1991b, Gil and Pérez, 1994]. The drill press in the simplified domain uses two types of drill bits, a *spot* drill and a *twist* drill. A spot drill makes a small spot on the surface of a part. This spot guides the movement of a twist drill, which makes a deeper hole.

The domain includes two classes of objects, PART and DRILL-BIT. The DRILL-BIT class consists of two subclasses, SPOT-DRILL and TWIST-DRILL. We use these classes to limit the scopes of variables in the operator description.

A *planning problem* is defined by a list of objects, an *initial state* I, and a *goal statement* G. The initial state is represented as a set of literals. The goal statement is a logical formula, which may contain conjunctions, disjunctions, negations, and universal and existential quantifiers. We show an example of a planning problem in Figure 2. In this example, we have five objects:



(a) Drill press.



(b) Classes of objects.

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>)

(c) Library of operators.

Figure 1: A simplified version of the PRODIGY Process-Planning Domain.

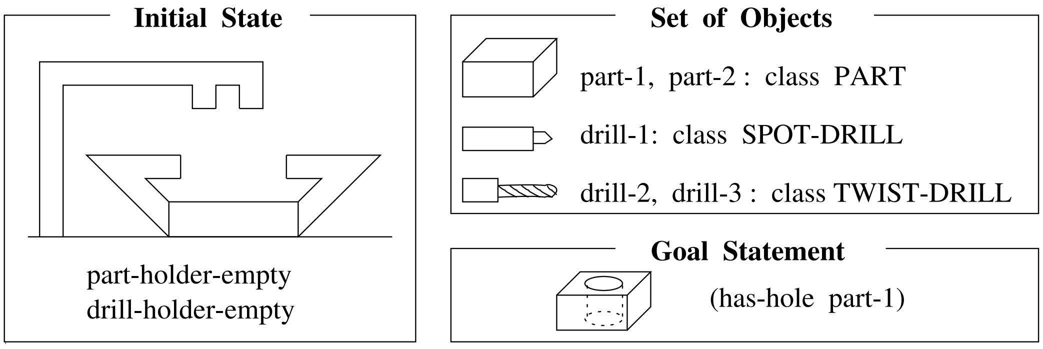


Figure 2: A problem in the simplified Process-Planning Domain.

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 to achieve 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 is called *correct*.

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. We may solve the problem 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)**."

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: The beginning of the partial-order solution to the problem in Figure 2.

3 Representation of Plans

Before presenting the planning algorithms used in PRODIGY, we describe the representation of incomplete plans generated by these algorithms during their search for a solution plan.

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

Different planning systems use different ways of representing incomplete plans. A plan may be a totally ordered sequence of operators (as in Strips [Fikes and Nilsson, 1971]) or 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 (for example, in NOLIMIT [Veloso, 1989]) or contain variables with codesignations (for example, in Tweak); the relations between operators and the goals they establish may be marked by causal links (for example, in NONLIN and SNLP).

In PRODIGY, an incomplete plan consists of two parts, the *head-plan* and *tail-plan* (see Figure 4). The tail-plan is built by a backward-chaining algorithm, which starts from the goal statement G and adds operators, one by one, to achieve goal literals and preconditions of tail-plan operators that are not satisfied in the current state.

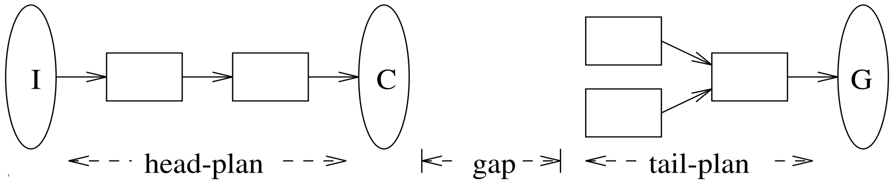


Figure 4: Representation of an incomplete plan.

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 values. The head-plan is generated by the execution-simulating algorithm described in Section 4. If the current incomplete plan is successfully modified to a correct solution to 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*. Note that, since the head-plan is a total-order plan that does not contain variables, the current state is *uniquely defined*. The backward-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.

We show an example of an incomplete plan in Figure 5. PRODIGY can construct this plan while solving the problem of making a spot in part-1. We can bridge the gap in this plan by a single operator, **put-drill-bit(drill-1)**.

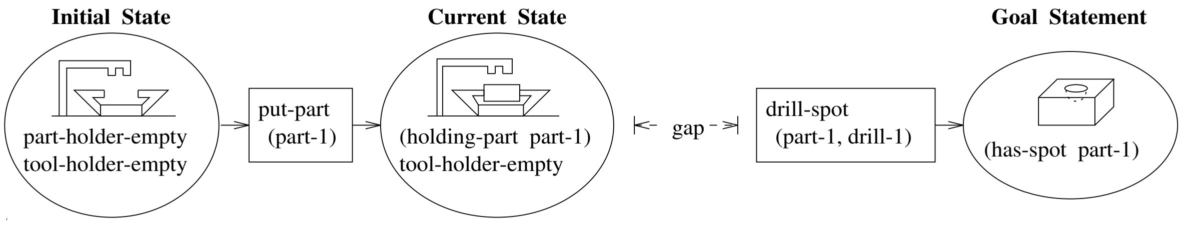


Figure 5: Example of an incomplete plan.

4 Simulating Plan Execution

We next describe an execution-simulating algorithm, the purpose of which is to enhance the backward-chaining planner with elements of forward chaining. The execution-simulator calls the backward-chainer to select operators relevant to the goal and simulates the application of these operators.

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 the current incomplete plan in one of two ways (see Figure 6). First, it can add an operator to the tail-plan (operator t in Figure 6). Modifications of the tail are handled by the backward-chaining planning algorithm. The backward-chainer views the current state C as the initial state. Almost any backward-chaining planner may be used as PRODIGY's backward-chainer. In Section 5, we describe two backward-chainers designed specifically for PRODIGY.

Second, PRODIGY can move some operator *op* from the tail to the head (operator *x* in Figure 6). The preconditions of *op* must be satisfied in the current state C. The operator *op* becomes the last operator of the head, and the current state is updated to account for the effects of *op*. PRODIGY usually has to select between several operators that can be applied to the current state.

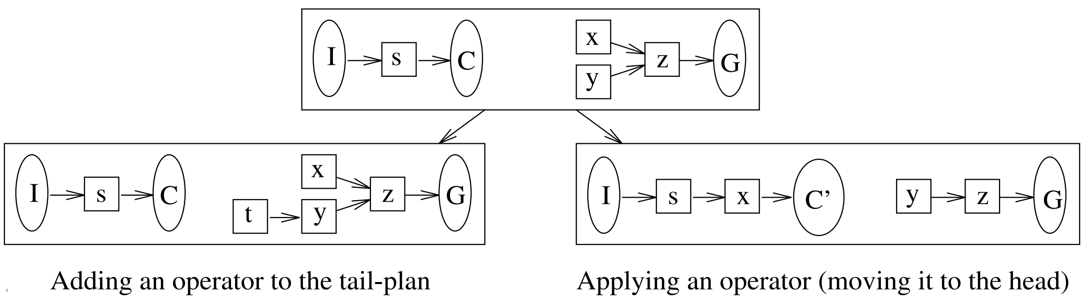


Figure 6: Modifying the current plan.

Intuitively, we 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 current state. Because of this analogy with the real-world changes, moving an operator from the tail to the end of the head is called the *application* of the operator. Note 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 can backtrack and consider an alternative execution sequence.

Note that, if we apply some operator *op* (that is, move *op* to the head-plan), then *op* will precede all other operators of the tail. Therefore, we can apply an operator only if it is *not* ordered after any other operator of the tail. For example, in the top plan of Figure 6, we can apply operator x or y, but not z. If the tail-plan is a total-order plan, we can apply only its first operator.

Moving an operator from the tail to the head is the only way of updating the head-plan. PRODIGY never inserts a new operator directly into the head. Thus, only goal-relevant operators are used in PRODIGY's forward chaining.

PRODIGY recognizes a plan as a solution to the problem when the head-plan achieves the goal; that is, the goal statement G is satisfied in C. PRODIGY may terminate after finding a solution or it may search for another plan.

We summarize the execution-simulating algorithm in Table 1. The places where PRODIGY chooses among several alternative modifications to the current plan are marked as *decision points*. Exploring different alternatives in decision points gives rise to different branches in the search space.

Prodigy4.0
1. If the goal statement G is satisfied in the current state C, then return *Head-Plan*.
2. Either
   (A) *Backward-Chainer* adds an operator to *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 *Prodigy4.0* on the resulting plan.

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

Table 1: Execution-simulating algorithm.

Note that, by construction, the head-plan is always valid. PRODIGY terminates when the goal statement G is satisfied in the current state C. Therefore, upon termination, the head-plan is always a valid plan that achieves G, that is, it is a correct solution to the planning problem. We conclude that PRODIGY is a sound planner, and its soundness does not depend on the backward-chaining algorithm.

The use of an unsound backward-chaining algorithm often improves the performance, because ensuring correctness in backward-chaining planning may be expensive, especially for partial-order plans. The comparative analysis of PRODIGY and SNLP [Veloso and Blythe, 1994] has demonstrated that an unsound backward-chainer makes PRODIGY much more efficient on a number of problems.

5 Backward-Chaining Algorithms

We now turn our attention to the backward-chaining techniques for constructing PRODIGY's tail-plan. We present two of the PRODIGY's backward-chainers, one of which operates with

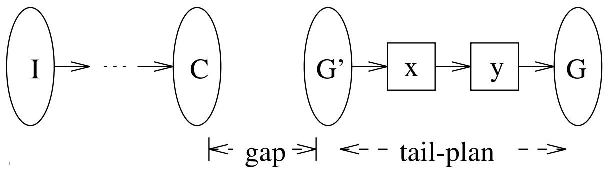


Figure 7: Representation of a total-order tail-plan.

total-order tail-plans, whereas the other is a partial-order planner.

Total-order backward-chainer

We describe an algorithm that represents the tail-plan as a total-order sequence of operators (see Figure 7). New operators are added to the beginning of the sequence. When the backward-chainer is called to modify the tail, it chooses an unachieved goal literal or operator precondition and adds a new operator to achieve this literal. A literal l is considered unachieved if

(1) l does not hold in the current state C

(2) and l is not established by any preceding operator of the tail-plan.

For example, given the plan in Figure 5, the backward-chainer may add the operator **put-drill-bit(drill-1)** to achieve the precondition (holding-tool drill1) of the **drill-spot** operator, thus generating the tail-plan "**put-drill-bit(drill-1), drill-spot(part-1,drill-1)**."

The set of unachieved goal literals and unachieved operator preconditions in the tail-plan is the *current goal* of planning. We denote this set by G'. For example, the current goal G' of the plan in Figure 5 includes the unachieved precondition (holding-tool drill1) of the **drill-spot** operator.

Initially, when the tail-plan is empty, the current goal comprises the goal literals that are not satisfied in the initial state: $G' = G - I$. When a new operator is added to the beginning of the tail-plan, the backward-chainer removes from G' the literals achieved by this operator and adds to G' the preconditions of the operator that are not satisfied in the current state C.

When the execution-simulator moves an operator from the beginning of the tail-plan to the end of the head-plan, the current goal G' must also be updated. To perform this update, the planner has to know the goal literals achieved by the moved operator. For this reason, the planner establishes links from every operator of the tail-plan to the literals achieved by the operator.

When inserting an operator into the tail-plan, the backward-chaining algorithm instantiates all variables in the operator description with specific values. Since PRODIGY's domain language allows complex constraints on the preconditions of operators, finding the set of possible instantiations may be a difficult problem. PRODIGY uses a constraint-based matching algorithm that considers all applicable instantiations of an operator [Wang, 1992].

For example, suppose that the planning goal is to drill a hole in part-1 and the backward-chaining algorithm adds the operator **drill-hole(<part>,<drill-bit>)** to achieve the goal literal has-hole(part-1) (see Figure 8). First, PRODIGY instantiates the variable <part> in the description of the **drill-hole** operator with the value part-1 from the goal literal. Then, the planner has to instantiate the other free variable, <drill-bit>. Since our domain contains two twist drills, drill-2 and drill-3, this variable has two possible instantiations, as shown in Figure 8.

We present a summary of the total-order backward-chaining algorithm in Table 2.

Partial-order backward-chainer

We now describe an algorithm that operates with tree-structured tail-plans (see Figure 9). The root of the tree is the goal statement G, the other nodes are operators, and the edges are the

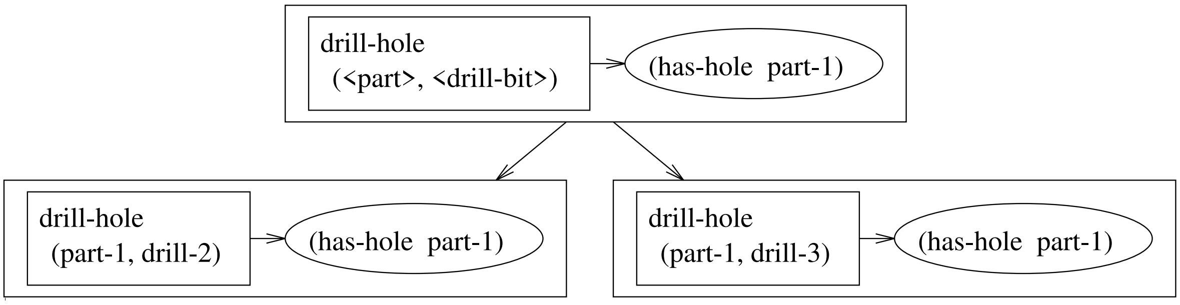


Figure 8: Instantiating a newly added operator.

Total-Order-Backward-Chainer
1. Pick a literal l from the current goal G'.
   Decision point: Choose one of the literals of G'.
2. Pick an operator *op* that achieves l.
   Decision point: Choose one of such operators.
3. Add *op* in the beginning of *Tail-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.
5. If the instantiated *op* achieves other literals of G', establish links to these literals as well.
6. Modify G': remove literals achieved by *op*
   and add the preconditions of *op* that are not satisfied in the current state C.

Table 2: Total-order backward-chaining algorithm.

ordering constraints. The planner adds operators to the tail-plan to achieve goal literals or preconditions of other tail-plan operators.

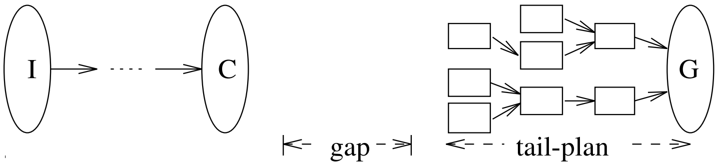


Figure 9: Tree-structured tail-plan.

A goal literal or operator precondition l in a tree-structured tail-plan is considered unachieved if

(1) l does not hold in the current state C

(2) and 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 literal l from the set G' of unachieved literals, adds a new operator *op* that achieves l, and establishes a link from *op* to l and the corresponding ordering constraint.

We present a summary of the backward-chaining algorithm in Table 3.

6 Conclusion

The PRODIGY planning system interleaves backward-chaining planning with simulation of plan execution. The backward-chainer is responsible for the goal-directed planning: it selects operators relevant to the goal of the planning problem. The simulator models the application of these goal-relevant operators to the initial state and computes the resulting current state.

When searching for a solution, PRODIGY operates with incomplete plans that consist of two parts: (1) the valid total-order head-plan, built by the execution-simulator and (2) the goal-directed tail-plan, constructed by the backward-chainer.

Partial-Order-Backward-Chainer

1. Pick an unachieved goal or precondition literal l, from G'.
   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; establish a link from *op* to l and the corresponding ordering constraint.
4. Instantiate the free variables of *op*.
   Decision point: Choose an instantiation for the variables of the operator.
5. Modify G': remove l and add preconditions of *op* that are not satisfied in C.

Table 3: Backward-chaining algorithm that generates tree-structured plans.

The efficiency of PRODIGY depends on its search space and on the order of expanding nodes of the search space. The search space is determined by the backward-chaining algorithm used by PRODIGY, whereas the order of expanding nodes depends on the branching decisions made in decision points. On each step, the planner decides which branch of the search space to explore first. PRODIGY's branching decisions are summarized in Figure 10.

A formal analysis of advantages and drawbacks of PRODIGY as compared to other planners is still an open research problem. However, multiple experiments have demonstrated PRODIGY's ability to solve efficiently a wide range of complex problems (see, for example, [Veloso, 1994] and [Gil, 1991a]).

We have recently described and compared different search strategies used in PRODIGY [Stone *et al.*, 1994]. We have also compared the PRODIGY planning algorithm with the SNLP partial-order planner [Veloso and Blythe, 1994] and identified situations in which PRODIGY is considerably more efficient than the planners that do not use execution simulation.

The results of comparing PRODIGY with other planners suggest that no single planning algorithm is universally superior for all planning domains. The effectiveness of a planning algorithm depends on a particular planning problem; different planning systems perform efficiently on different problems.

We conclude with a summary of important features of the PRODIGY planning and learning system.

Rich domain language. PRODIGY's language for operator representation includes disjunctive preconditions, universal and existential quantification of variables, and conditional and functional effects. The execution-simulator, which maintains the description of the current state of the world, allows us to use this powerful operator representation without

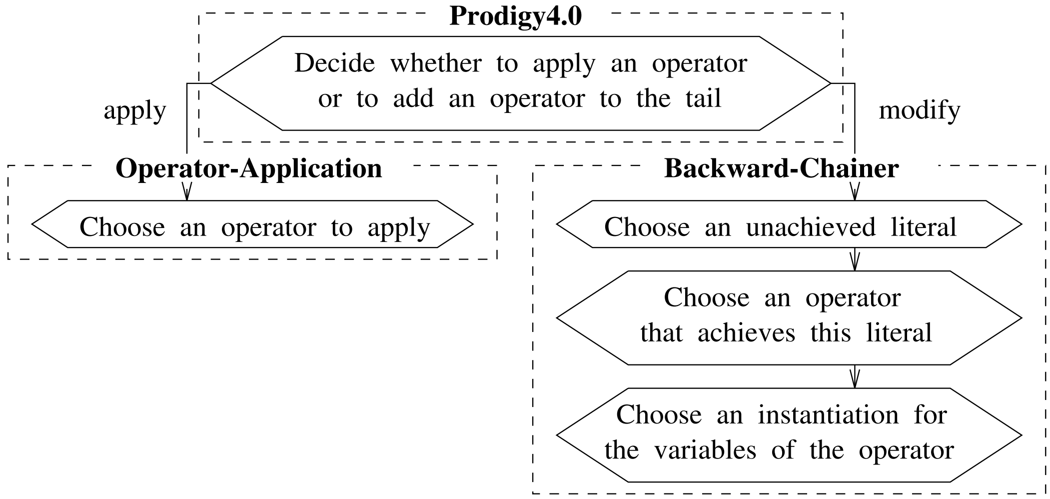


Figure 10: Branching decisions.

compromising the efficiency of planning.

Combining execution simulation and backward chaining. The execution-simulator enables PRODIGY to use information about the updated world state in planning and learning. The application of operators often considerably reduces the size of the tail-plan, thus improving the efficiency of the backward-chaining algorithm.

Learning opportunities. PRODIGY uses several learning algorithms, which improve the efficiency of the planner. The learning algorithms use information about the current state to identify the reasons for local and global planning successes and failures. Partial-order constraints of the tail-plan enable learners to determine which aspects of the operator ordering in the head-plan are relevant to the solution.

Acknowledgements

PRODIGY2.0 was designed and implemented by Steven Minton, Jaime Carbonell, Craig Knoblock, and Dan Kuokka. The next version, NOLIMIT, was created by Manuela Veloso and Daniel Borrajo. Based on these two algorithms, Jim Blythe, Xuemei Wang, Dan Kahn, and Alicia Pérez implemented the PRODIGY4.0 system.

References

[Carbonell *et al.*, 1992] Jaime G. Carbonell, Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Pérez, Scott Reilly, Manuela M. Veloso, and Xuemei Wang. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, 1992.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Gil and Pérez, 1994] Yolanda Gil and Alicia Pérez. Applying a general-purpose planning and learning architectures to process planning. In *Proceedigs of the AAAI 1994 Fall Symposium on Planning and Learning*, pages 48–52, 1994.

[Gil, 1991a] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Technical Report CMU-CS-92-175.

[Gil, 1991b] Yolanda Gil. A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, 1991.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68, 1994.

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

[Minton *et al.*, 1989] Steven Minton, Dan R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1988. Technical Report CMU-CS-88-133.

[Stone *et al.*, 1994] Peter Stone, Manuela M. Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169, 1994.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 888–900, 1977.

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

[Veloso and Borrajo, 1994] Manuela M. Veloso and Daniel Borrajo. Learning strategy knowledge incrementally. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 484–490, New Orleans, LA, 1994.

[Veloso *et al.*, 1995] Manuela M. Veloso, Jaime G. Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

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

[Veloso, 1994] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, 1994.

[Wang, 1992] Xuemei Wang. Constrained-based efficient matching in PRODIGY. Technical Report CMU-CS-92-128, School of Computer Science, Carnegie Mellon University, 1992.

[Wang, 1994] Xuemei Wang. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 335–340, 1994.