Prodigy Planning Algorithm

Eugene Fink Manuela Veloso

March 1994

CMU-CS-94-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a formal description of the planning algorithm used in the Prodigy4.0 system. The algorithm is based on an interesting combination of backward-chaining planning and simulation of plan execution. The backward-chainer selects goal-relevant operators, and then Prodigy simulates their application to the current state of the world. The system can use different backward-chaining procedures, some of which are presented in the paper.

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 U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes.

Keywords: planning, problem solving, Prodigy

1 Introduction

Prodigy is an integrated planning and learning system. The system includes not only a planning algorithm but also procedures for learning and case-based reasoning, which greatly increase the efficiency of the planner. For example, Prodigy is able to learn control rules [Minton, 1988], conduct experiments to acquire new knowledge [Gil, 1992], generate abstraction hierarchies [Knoblock, 1993], and use analogical reasoning to recognize and exploit similarities between planning problems [Veloso, 1992].

Prodigy's core, the planning algorithm itself, has been improved over the years. The old algorithm, Prodigy2.0 [Minton *et al.*, 1989], was succeeded by NoLimit [Veloso, 1989] and then by Prodigy4.0 [Carbonell *et al.*, 1992]. All versions of Prodigy were developed by members of the Prodigy research project at Carnegie Mellon University. The authors of the system are listed in the acknowledgements section.

Strictly speaking, Prodigy4.0 is not a single planner, but a *family* of closely related planning procedures. These procedures are based on a combination of backward-chaining with a simulation of plan execution, similar to forward-chaining planning. Prodigy4.0 can use different backward-chaining procedures, some of which are presented in the paper. Prodigy's execution-simulator searches among states of the world that can be achieved from the initial state by applying different operators. However, unlike usual forward-chainers, the simulator uses only operators selected by a backward-chaining algorithm as relevant to the goal.

The goal of our paper is to give a formal description of the Prodigy planning algorithm. We show how Prodigy4.0 represents the plans, describe its search space, and give an overview of different planners of the Prodigy family.

Prodigy4.0 consists of two parts: a simulator of the plan execution and a backward-chaining procedure. The backward-chainer is responsible for goal-directed reasoning, while the execution-simulator enhances the goal-directed search with elements of forward-chaining. These two parts of Prodigy are described in the two large sections of the paper: the execution-simulator in Section 3 and a family of backward-chainers in Section 4.

2 Definitions

A *planning domain* is defined by a library of operators. Prodigy's language for describing operators is based on the Strips domain language [Fikes and Nilsson, 1971], extended to express disjunctive preconditions, universal quantification, functions, and conditional effects [Carbonell *et al.*, 1992]. An example of a planning domain, a version of the Blocks World, is shown in Table 1. The operators in this example contain variables, which refer to blocks and hands.

Prodigy usually replaces variables of an operator with particular constants before inserting the operator into a plan. The instantiation is performed by a complex matching algorithm, which considers all applicable instances of an operator and does not compromise the completeness of planning [Wang, 1992]. A description of this matching algorithm beyond the scope of the paper.

A *planning problem* is defined by an *initial state* I and a *goal* G, represented as sets of literals. A solution of a planning problem is a sequence of operators that can be applied to

unstack (hand, x, y)	**stack (hand, x, y)**	**pick-up (hand, x)**	**put-down (hand, x)**
Pre: on(x,y) clear(x) empty(hand)	*Pre:* in-hand(hand,x) clear(y)	*Pre:* on-table(x) clear(x) empty(hand)	*Pre:* in-hand(hand,x)
Add: in-hand(hand,x) clear(y)	*Add:* on(x,y) clear(x) empty(hand)	*Add:* in-hand(hand,x)	*Add:* on-table(x) clear(x) empty(hand)
Del: on(x,y) clear(x) empty(hand)	*Del:* in-hand(hand,x) clear(y)	*Del:* on-table(x) clear(x) empty(hand)	*Del:* in-hand(hand,x)

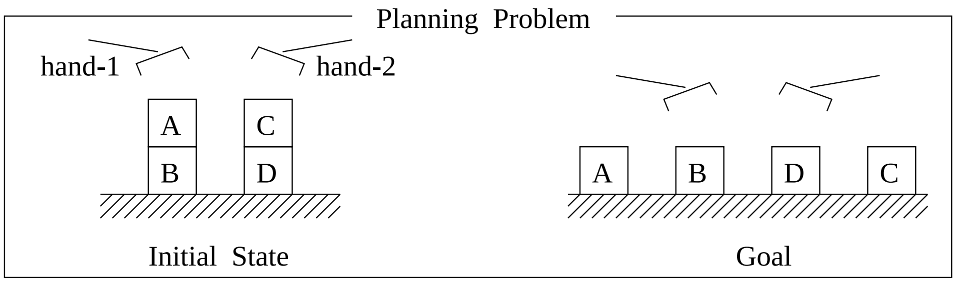

Table 1: The Blocks World domain (with two hands)

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 G is called *correct*.

For example, consider the planning problem in Table 1. The plan "*unstack*(hand-1,A), *put-down*(hand-1,A)" is valid, since it can be applied to the initial state. However, this plan does not achieve the goal, and hence it is *not* correct. The problem may be solved by the plan "*unstack*(hand-1,A), *put-down*(hand-1,A), *unstack*(hand-2,C), *put-down*(hand-2,C)," which *is* correct.

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 problem in Table 1 may be solved by the following partial-order plan:

unstack (hand-1, A) → *put-down* (hand-1, A)
unstack (hand-2, C) → *put-down* (hand-2, C)

3 Simulating plan execution

The purpose of the algorithm described in this section is to enhance a backward-chaining planner with elements of forward-chaining. This algorithm calls a back-chainer to select operators relevant to the goal and simulates the application of these operators. Some of Prodigy's back-chainers will be described in Section 4.

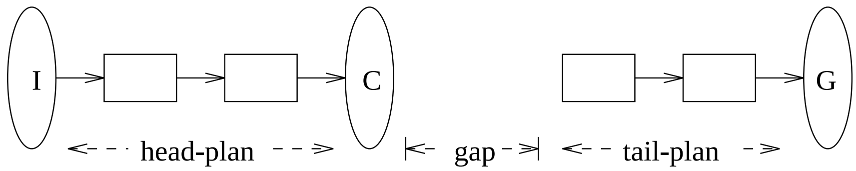

Figure 1: Representation of an incomplete plan

3.1 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 an ordering constraint, or instantiating a variable. 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 plan corresponds to expanding a node. The branching factor of search is determined by the number of possible modifications of the current plan.

An incomplete plan may be represented in many different ways. It may be a total-order sequence of operators (as in Strips [Fikes and Nilsson, 1971]) or a partial-order plan (as in Tweak [Chapman, 1987], Non-Lin [Tate, 1977], and SNLP [McAllester and Rosenblitt, 1991]); the operators of the plan may be instantiated (e.g. in NoLimit) or contain variables with codesignations (e.g. in Tweak); the relations between operators and the goals they establish may be marked by casual links (e.g. in Non-Lin and SNLP).

In Prodigy, an incomplete plan consists of two parts, the *head-plan* and the *tail-plan* (see Figure 1). The tail-plan is built by a backward-chaining algorithm, which starts from the goal G and adds operators, one by one, to achieve goal literals and preconditions of other operators. Prodigy may use different backward-chaining algorithms. The tail-plan may be total-order or partial-order, depending on a particular backward-chainer.

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 replaced with specific constants. The head-plan is built 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 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 all preconditions of the tail-plan are satisfied in C, then the tail-plan may be executed immediately after the head, and thus the head and tail together make a solution of the planning problem. If some preconditions of the tail-plan are not satisfied in C, then there is a "gap" between the head and tail. The purpose of planning is to bridge this gap.

Figure 2 shows an example of an incomplete plan with a partial-order tail. This plan can be constructed by Prodigy while solving the problem in Table 1. The gap in this plan can be bridged by a single operator, *unstack*(hand-2,C).

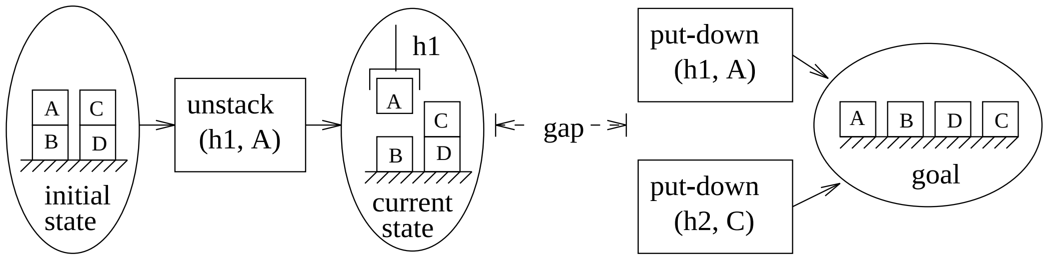

Figure 2: Example of an incomplete plan ("h1" stands for "hand-1")

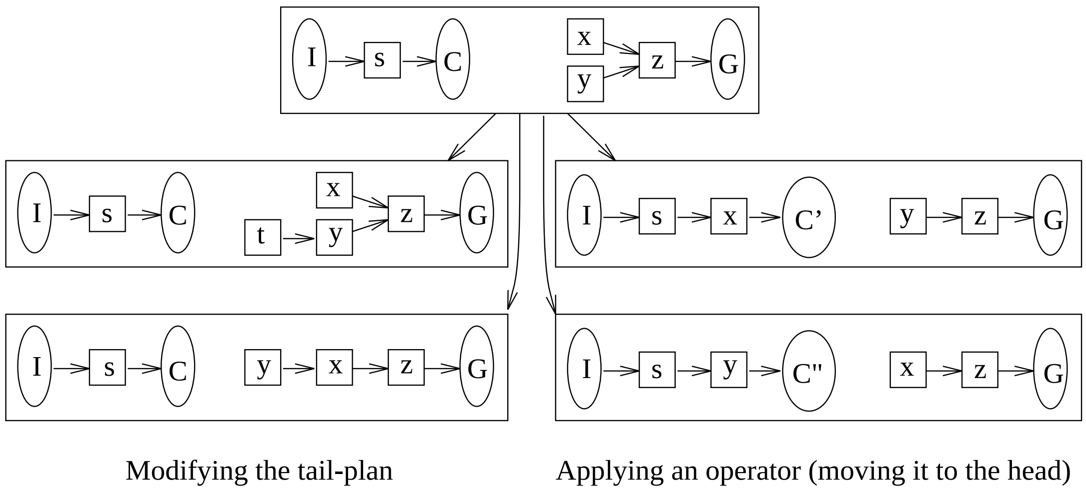

Figure 3: Modifying the current plan

3.2 Prodigy's top-level algorithm: Execution-simulator

Given an initial state I and a goal 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 3). First, it can modify the tail-plan. Modifications of the tail are handled by a separate planning procedure, called *Back-Chainer*. For example, *Back-Chainer* may add a new operator to the tail or impose an ordering constraint. This procedure views the current state C as the initial state. Almost any backward-chaining planner may be used as Prodigy's *Back-Chainer*. Two back-chainers designed specifically for Prodigy are described in Section 4.

Second, Prodigy can move some operator op from the tail to the head (see Figure 3). The preconditions of op must 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.

For example, the operator *put-down*(hand-1,A) in Figure 2 can be moved to the head. On the other hand, *put-down*(hand-2,C) cannot be moved, because its preconditions are not satisfied in the current state (Block C is not in hand-2).

Intuitively, we may think 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 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 solution.

Notice that if we apply some operator *op* (i.e. we 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 3, we can apply operator x or y, but not z. If the tail-plan is total-order, we can apply only its first operator.

If operators in the tail-plan contain variables, then we must instantiate *op* before applying it. The instantiation may be performed by Prodigy's matching algorithm. All possible instantiations must be considered to insure completeness.

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 of the problem if the head-plan achieves the goal G, i.e. all goal literals are *True* in C. Prodigy may terminate after finding a solution, or it may search for a better plan.

Table 2 summarizes the execution-simulating algorithm. The places where Prodigy chooses among several alternative modifications of the current plan are marked as *backtracking* points.

3.3 Soundness and completeness

Below we discuss the formal properties of Prodigy, *soundness* and *completeness*, and their connection with the corresponding properties of *Back-Chainer*. We show that Prodigy is always sound, while its completeness depends on the search strategy used by Prodigy and on the *Back-Chainer* procedure.

Soundness: *A plan found by Prodigy for a given problem is always a correct solution of the problem.*

Sketch of the proof. By construction, the head-plan is a plan that can be validly applied to the initial state. Prodigy terminates when the goal G is satisfied in the current state C, achieved by applying the head-plan. Therefore, upon termination, the head-plan is a correct plan that achieves G. $\square$

The theorem shows that Prodigy *is* a sound planner even if *Back-Chainer* is not sound. This property enables us to use unsound back-chaining algorithms. The use of unsound algorithms sometimes improves performance, because insuring correctness in backward-chaining

Prodigy

1. If the goal G is satisfied in the current state C, then return *Head-Plan*.
2. Either
   (A) *Back-Chainer* modifies *Tail-Plan*, or
   (B) *Operator-Application* moves an operator from *Tail-Plan* to *Head-Plan*.
   Backtracking point: both alternatives must be considered.
3. Recursively call *Prodigy* on the resulting plan.

Operator-Application

1. Pick an operator *op* in *Tail-Plan* such that
   (A) there is no operator in *Tail-Plan* ordered before *op*, and
   (B) the preconditions of *op* are satisfied in the current state C.
   Backtracking point: all such operators must be considered.
2. Instantiate *op* if necessary.
3. Move *op* to the end of *Head-Plan* and update the current state C.

Table 2: Execution-simulating algorithm

planning may be expensive, especially for partial-order plans. We describe in the next section two unsound partial-order back-chainers used in Prodigy.

Next we analyze completeness of Prodigy. A planner is said to be *complete* if it is able to find a solution for any solvable problem.

Completeness: *If* Back-Chainer *is complete, and if Prodigy explores* all *branches of the search space (e.g. by the breadth-first search or iterative deepening), then Prodigy is a complete planner.*

Sketch of the proof. If we remove all branches of Prodigy's search space where operators are moved to the head, then Prodigy can only modify the tail-plan (by calling *Back-Chainer*), and thus Prodigy's space becomes identical to the search space of *Back-Chainer*. We conclude that *Back-Chainer*'s space is a *subset* of the entire space of Prodigy. If a planning problem has a solution, then the space of a complete *Back-Chainer* contains a solution plan, and therefore Prodigy's space also contains this solution plan. By successively applying all operators of this plan, Prodigy will find the solution head-plan and terminate. □

The proof shows that Prodigy's search space *contains* the space of *Back-Chainer*, and that the path to a solution in Prodigy's space is longer than *Back-Chainer*'s path to the same solution. Thus, the breadth-first search by Prodigy is less efficient than the search by *Back-Chainer*. Prodigy becomes efficient only when it uses the depth-first search.

4 Back-chaining procedures

We now turn our attention to *Back-Chainer* procedures used in Prodigy. We present two of the Prodigy's *Back-Chainers*, one of which operates with total-order tail-plans, while the other is a partial-order planner. Recall that the completeness of Prodigy depends on the *Back-Chainer*. On the other hand, the soundness of back-chaining procedures is not a concern.

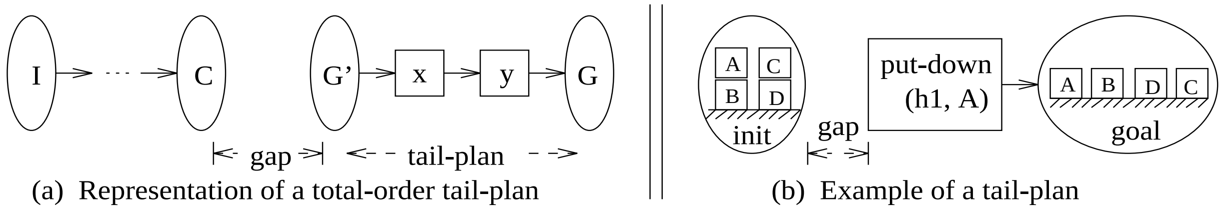

Figure 4: Total-oreder tail plan

4.1 Total-order back-chainer

This algorithm represents the tail as a total-order sequence of operators (see Figure 4a). New operators are added to the beginning of the sequence. When this *Back-Chainer* is called to modify the tail, it chooses an unachieved precondition or goal literal and adds a new operator that achieves this literal. A literal l is considered *unachieved* if

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

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

For example, given the plan shown in Figure 4b, the *Back-Chainer* may add the operator *put-down*(hand-2,C) to achieve the goal literal *on-table*(C), thus generating the tail-plan "*put-down*(hand-2,C), *put-down*(hand-1,A)."

The set of unachieved goal literals and unachieved operator preconditions in the tail-plan may be viewed as the *current goal* of planning. Let us denote this set by G'. For example, the current goal G' of the plan in Figure 4b includes the unachieved goal literal *on-table*(C) and the precondition *in-hand*(hand-1,A) of *put-down*(hand-1,A).

Initially, when the tail-plan is empty, the current goal consists of the goal literals that are not satisfied in the initial state: $G' = G - I$. When a new operator *op* is added in the beginning of the tail-plan, *Back-Chainer* removes from G' the literals achieved by *op* and adds to G' the preconditions of *op* that are not satisfied in the current state C. If Prodigy moves *op* from the beginning of the tail-plan to the end of the head-plan, G' also must be updated. To perform this update, we have to know the goal literals achieved by *op*. For this reason, we establish links from every operator of the tail-plan to the literals achieved by the operator.

Table 3 summarizes our total-order back-chaining algorithm. By itself, this back-chaining algorithm is not complete. However, in combination with Prodigy's execution-simulator it makes a complete planner.

4.2 Partial-order back-chainers

Tree-structured plans. A simple partial-order *Back-Chainer* operates with plans that are organized as trees (see Figure 5). The root of the tree is the goal G, other nodes are operators, and edges are ordering constraints. Each goal literal is linked to the operator of the tail-plan that was selected to achieve it.

A goal literal or precondition l in a tree-like tail-plan is *unachieved* if

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

Total-Order-Back-Chainer
1. Pick a literal l from G'.
   Backtracking point: all literals of G' must be considered.
2. Pick an operator op that achieves l.
   Backtracking point: all such operators must be considered.
3. Add op in the beginning of *Tail-Plan.*
   Establish a link from op to l.
   If op achieves other literals of G', establish links to these literals as well.
4. Modify G': remove literals achieved by op and
   add preconditions of op that are not satisfied in C.

Table 3: Total-order back-chaining algorithm

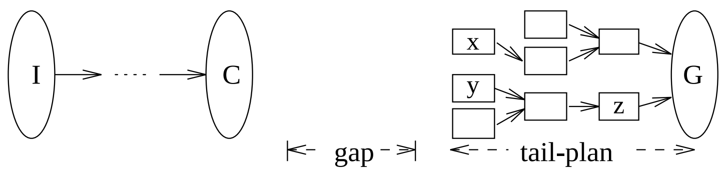

Figure 5: Tree-structured tail-plan

(2) l is not linked with any operator.
When the back-chaining procedure 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. The summary of the algorithm is shown in Table 4. One may verify that *this back-chainer is complete.*

The backtracking 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 tree-plan.

An enhanced back-chainer. Sometimes, a precondition of an operator in a partial-order tail-plan may be achieved by another operator of the tail. For example, operator x in Figure 5 may establish some precondition of y. Then we may achieve this precondition of y by establishing a link from x to y rather than adding a new operator. When establishing new

Partial-Order-Back-Chainer
1. Pick a literal l from G'.
   Backtracking: all such literals must be considered.
2. Pick an operator op that achieves l.
   Backtracking point: all such operators must be considered.
3. Add op to the plan and establish a link from op to l.
4. Modify G': remove l and add preconditions of op that are not satisfied in C.

Table 4: Back-chainer generating tree-structured plans

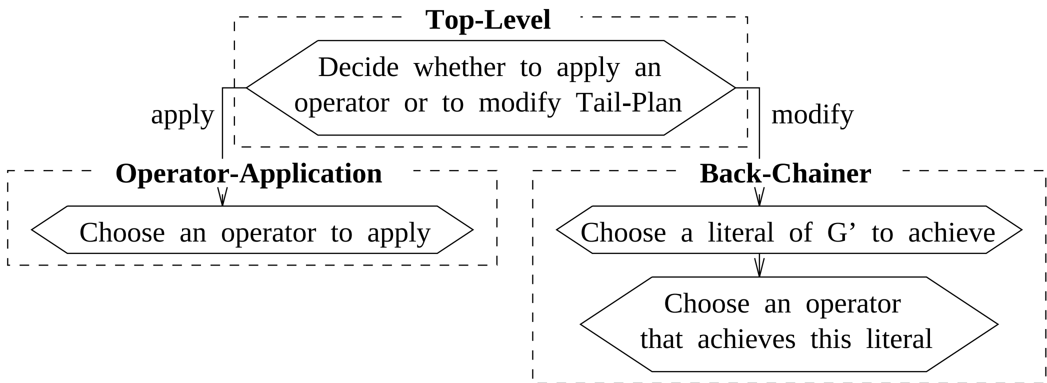

Figure 6: Branching decisions

links, we must make sure that links do not form loops. For example, we cannot establish a link from z to y (Figure 5). We can readily extend our partial-order back-chainer to the procedure that not only adds new operators but also establishes new links between old operators of the tail-plan.

5 Conclusion

The Prodigy planning algorithm interleaves backward-chaining planning with the simulation of plan execution. The back-chainer is responsible for the goal-directed planning: it provides the execution-simulator with operators relevant to the goal of the planning problem. The simulator models possible ways of applying goal-relevant operators to the initial state of the world and computes the resulting current state. The execution-simulator insures the soundness of planning, while the back-chainer is responsible for the completeness.

Incomplete plans generated by Prodigy while searching for a solution 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 back-chainer. Different representations of the tail-plan give rise to different back-chaining procedures.

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 back-chaining procedure used by Prodigy, while the order of expanding nodes depends on the branching decisions made in backtracking points. On each step, the planner decides which branch of the search space to explore first. The decisions made by Prodigy in every backtracking point are summarized in Figure 6.

The paper does not address the efficiency of the Prodigy system. 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 efficiently solve a wide range of complex problems (see, for example, [Veloso, 1992] and [Gil, 1992]). Below we list some advantages of Prodigy's planning algorithm.

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, allows us to use this powerful operator representation.

Learning opportunities. Prodigy uses several learning procedures, which improve the efficiency of the planner. The information about the current state is used at a learning phase 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 and Dan Kahn developed the Prodigy4.0 system. (Jim Blythe led this development team.) Prodigy4.0 was the result of many design discussions with the other members of the Prodigy research group, including Alicia Pérez, Oren Etzioni, and the authors of the earlier versions of Prodigy.

References

[Carbonell *et al.*, 1992] Jaime Carbonell, Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Pérez, Scott Reilly, Manuela Veloso, and Xuemei Wang. *Prodigy4.0: the manual and tutorial.* Carnegie Mellon, Computer Sci., 1992. Tech. Rep. CMU-CS-92-150.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32, pages 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, pages 189–208, 1971.

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon, Computer Sci., 1992. Tech. Rep. CMU-CS-92-175.

[Knoblock, 1993] Craig Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 1993, in press.

[McAllester and Rosenblitt, 1991] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of AAAI*, pages 634-639, 1991.

[Minton *et al.*, 1989] Steven Minton, Craig Knoblock, Dan Kuokka, Yolanda Gil, Robert Joseph, and Jaime Carbonell. *Prodigy2.0: the manual and tutorial.* Carnegie Mellon, Computer Sci. Tech. Rep. CMU-CS-89-146.

[Minton, 1988] Steven Minton. *Learning effective search control knowledge: an explanation-based approach.* Kluwer Academic Publishers, Boston, MA, 1988.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of IJCAI*, pages 888-893, 1977.

[Veloso, 1989] Manuela Veloso. *Nonlinear problem solving using intelligent casual-commitment.* Carnegie Mellon, Computer Sci., 1988. Tech. Rep. CMU-CS-89-210.

[Veloso, 1992] Manuela Veloso. *Learning by analogical reasoning in general problem solving.* PhD thesis, Carnegie Mellon, Computer Sci., 1992. Tech. Rep. CMU-CS-92-174.

[Wang, 1992] Xuemei Wang. *Constraint-based efficient matching in Prodigy.* Carnegie Mellon, Computer Sci., 1992. Tech. Rep. CMU-CS-92-128.