Prodigy Bidirectional Planning

Eugene Fink
Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
e.fink@cs.cmu.edu
www.cs.cmu.edu/∼eugene

Jim Blythe
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
blythe@isi.edu
www.isi.edu/∼blythe

Abstract

The PRODIGY system is based on *bidirectional planning*, which is a combination of goal-directed reasoning with simulated execution. Researchers have implemented a series of planners that utilize this search strategy, and demonstrated that it is an efficient technique, a fair match for other successful planners; however, they have provided few formal results on the common principles underlying the developed algorithms.

We formalize bidirectional planning, elucidate some techniques for improving its efficiency, and show how different strategies for controlling search complexity give rise to different versions of PRODIGY. In particular, we demonstrate that PRODIGY is incomplete and discuss advantages and drawbacks of its incompleteness. We then develop a complete bidirectional planner and compare it experimentally with PRODIGY. We show that it is almost as fast as PRODIGY and solves a wider range of problems.

1 Introduction

Newell and Simon [1961; 1972] invented *means-ends analysis* during their work on the General Problem Solver (GPS), back in the early days of artificial intelligence. Their technique combined goal-directed reasoning with forward chaining from the initial state. The authors of later planning systems [Fikes and Nilsson, 1971; Warren, 1974; Tate, 1977] gradually abandoned forward search and began to rely exclusively on backward chaining.

Researchers studied several types of backward chainers [Minton *et al.*, 1994] and showed that *least commitment* improves the efficiency of goal-directed reasoning, which gave rise to TWEAK [Chapman, 1987], ABTWEAK [Yang *et al.*, 1996], SNLP [McAllester and Rosenblitt, 1991], UCPOP [Penberthy and Weld, 1992; Weld, 1994], and other least-commitment planners.

Meanwhile, PRODIGY researchers extended means-ends analysis and developed a family of planners based on the combination of goal-directed backward chaining with simulation of plan execution [Veloso *et al.*, 1995]. The underlying strategy is a special case of bidirectional search [Pohl, 1971]. It gave rise to several versions of the PRODIGY system, including PRODIGY1, PRODIGY2, NOLIMIT, PRODIGY4, and FLECS.

version	year	authors
PRODIGY1	1986	Minton and Carbonell
PRODIGY2	1989	Carbonell, Minton, Knoblock, and Kuokka
NOLIMIT	1990	Veloso and Borrajo
PRODIGY4	1992	Blythe, Wang, Veloso, Kahn, Pérez, and Gil
FLECS	1994	Veloso and Stone

Table 1: Main versions of the PRODIGY architecture. The work on this planning architecture continued for over ten years and gave rise to a series of novel search strategies.

The developed planners keep track of the world state, which results from executing parts of the constructed plan, and use the state to guide the goal-directed reasoning. Least commitment proved ineffective for bidirectional search, and Veloso developed an alternative strategy, called *casual commitment*, based on instantiating all variables as early as possible [Veloso, 1994].

Experiments have shown that bidirectional planning is an efficient technique, a fair match for least-commitment systems and other successful planners. Moreover, the PRODIGY architecture has proved a valuable tool for the development of speed-up learning techniques.

Although researchers have applied bidirectional planners to a variety of domains, and conducted empirical studies of their performance, they have provided few formal results on the common principles underlying these planners. To date, all bidirectional planners have been incomplete, and researchers have not studied reasons for their incompleteness.

The purpose of the described work is to formalize the principles of bidirectional planning. We analyze the bidirectional technique, describe general heuristics for improving its efficiency, and elucidate differences among the main versions of PRODIGY. We then identify the two specific reasons for the incompleteness of means-ends analysis and develop the first complete bidirectional planner, by extending PRODIGY search. The planner supports a rich domain language, which allows the use of conditional effects and complex precondition expressions [Carbonell *et al.*, 1992].

We review the past work on PRODIGY (Section 1.1) and describe the foundations of the developed techniques (Section 2), as well as the main extensions to the basic search engine (Sections 3 and 4). We then present a complete bidirectional planner and show that it is almost as efficient as PRODIGY and that it solves more problems (Section 5).

1.1 PRODIGY history

The PRODIGY system went through several stages of development and gradually evolved into an advanced architecture that supports a variety of search and learning techniques. Its history (see Table 1) began circa 1986, when Steven Minton and Jaime Carbonell implemented PRODIGY1, which became a testbed for their work on control rules [Minton, 1988; Minton *et al.*, 1989a].

Steven Minton, Jaime Carbonell, Craig Knoblock, and Dan Kuokka used PRODIGY1 as a prototype in their work on PRODIGY2 [Carbonell *et al.*, 1991], which supported an advanced language for describing domains [Minton *et al.*, 1989b]. They demonstrated the system's effectiveness in scheduling machine-shop operations [Gil, 1991; Gil and Pérez, 1994], planning a robot's actions in an extended STRIPS world [Minton, 1988], and solving a variety of smaller problems.

Manuela Veloso [1989] and Daniel Borrajo developed the next bidirectional planner, called NOLIMIT, which significantly differed from its predecessors. In particular, they added new decision points and introduced object types for specifying possible values of variables. Veloso demonstrated the effectiveness of NOLIMIT on the previously designed PRODIGY domains, as well as on transportation problems.

Jim Blythe, Mei Wang, Manuela Veloso, Dan Kahn, Alicia Pérez, and Yolanda Gil developed a collection of techniques for enhancing bidirectional planners and built PRODIGY4 [Carbonell *et al.*, 1992]. They provided an efficient technique for instantiating operators [Wang, 1992], extended the use of inference rules, and designed advanced data structures for the low-level implementation. They also designed a user interface and tools for adding new learning mechanisms.

Manuela Veloso and Peter Stone [1995] implemented the FLECS planner, an extension to PRODIGY4 that included an additional decision point and new search strategies, and showed that their strategies often improved the efficiency.

The PRODIGY architecture provided ample opportunities for speed-up learning, and researchers used it to develop a variety of techniques for automated efficiency improvement. Minton [1988] designed the first learning module for PRODIGY, which automatically generated control rules. He demonstrated effectiveness of integrating PRODIGY search with learning, which stimulated work on other speed-up techniques.

In particular, researchers designed modules for explanation-based learning [Etzioni, 1990; Etzioni, 1993; Pérez and Etzioni, 1992], inductive generation of control rules [Veloso and Borrajo, 1994; Borrajo and Veloso, 1996], abstraction search [Knoblock, 1993], and analogical reuse of planning episodes [Carbonell, 1983; Veloso and Carbonell, 1990; Veloso and Carbonell, 1993a; Veloso and Carbonell, 1993b; Veloso, 1994]. They also investigated techniques for improving the quality of plans [Pérez and Carbonell, 1993; Pérez, 1995], learning unknown properties of a domain [Gil, 1992; Carbonell and Gil, 1990; Wang, 1994; Wang, 1996], and collaborating with the human user [Joseph, 1992; Stone and Veloso, 1996; Cox and Veloso, 1997a; Cox and Veloso, 1997b; Veloso *et al.*, 1997]. The reader may find a summary of PRODIGY learning techniques in the review papers by Carbonell *et al.* [1990] and Veloso *et al.* [1995].

1.2 Advantages and drawbacks

The PRODIGY architecture is based on two major design decisions. First, it combines backward chaining with simulated plan execution. Second, it fully instantiates operators in early stages of search, whereas most classical planners delay the instantiation.

The backward-chaining procedure selects operators relevant to the goal, instantiates them, and arranges them into a partial-order plan. The forward chainer simulates the execution of these operators and gradually constructs a total-order sequence of operators. The system keeps track of the simulated world state that would result from executing this sequence. It utilizes the simulated state in selecting operators and their instantiations, which improves the effectiveness of goal-directed reasoning. In addition, learning modules use the state to identify reasons for successes and failures.

Since PRODIGY uses fully instantiated operators, it efficiently handles a powerful domain language. In particular, it supports disjunctive and quantified preconditions, conditional effects, and arbitrary constraints on the values of operator variables [Carbonell *et al.*, 1992]. The planner utilizes the world state in choosing appropriate instantiations.

On the flip side, full instantiation leads to a large branching factor, which results in gross inefficiency of breadth-first search. The planner uses depth-first search and relies on heuristics for selecting branches of the search space, which usually leads to suboptimal plans. If the heuristics turn out misleading, the planner may fail to find a solution.

A formal comparison of PRODIGY and other planners is still an open problem; however, multiple empirical studies confirmed that PRODIGY search is an efficient strategy [Stone *et al.*, 1994]. Experiments also revealed that PRODIGY and backward chainers perform well in different domains. Some tasks are more suitable for execution simulation, whereas others require backward chaining. Veloso and Blythe [1994] identified some domain properties that determine which of the two strategies is more effective.

Kambhampati and Srivastava [1996a; 1996b] investigated common principles underlying PRODIGY and least-commitment search. They developed a framework that generalizes these two types of goal-directed reasoning and combined them with direct forward search. They implemented the Universal Classical Planner, which can use all these strategies; however, the resulting algorithm has many decision points, which give rise to an impractically large search space. The main open problem is the development of heuristics that would effectively use the flexibility of the Universal Classical Planner.

Blum and Furst [1997] constructed GRAPHPLAN, which uses the world state in a different way. It utilizes propagation of constraints from the initial state to identify operators whose preconditions cannot be satisfied. The planner discards these operators and uses backward chaining with the remaining operators. GRAPHPLAN performs forward constraint propagation prior to its search for a solution. Unlike PRODIGY, it does not use forward search from the initial state.

The relative performance of PRODIGY and GRAPHPLAN varies across domains. The GRAPHPLAN algorithm has to generate all possible instantiations of all operators before searching for a solution, which often causes a combinatorial explosion in large-scale domains. On the other hand, GRAPHPLAN is faster in small-scale domains that require extensive search.

Researchers applied PRODIGY to robot navigation and discovered that its execution simulation is useful for interleaving search with real execution. In particular, Blythe and Reilly [1993a; 1993b] explored techniques for planning the routes of a household robot in a simulated environment. Stone and Veloso [1996] constructed a mechanism for user-guided interleaving of planning and execution.

Haigh and Veloso [1997; 1998a; 1998b] built a system for navigating XAVIER, a real robot at Carnegie Mellon University. Haigh [1998] integrated this system with XAVIER's low-level control and demonstrated its effectiveness in guiding the robot's actions. It begins the real-world execution before completing the search for a solution, which involves the risk of guiding the robot into an inescapable trap. To avoid such traps, Haigh and Veloso restricted their system to domains with reversible actions.

2 Search engine

We describe the basics of bidirectional planning, including the results of joint work with Veloso on the principles underlying PRODIGY [Fink and Veloso, 1996]. We present the PRODIGY domain language (Section 2.1), encoding of intermediate incomplete plans (Section 2.2), and algorithm that combines backward chaining with execution simulation (Sections 2.3 and 2.4). Then, we discuss differences among the versions of PRODIGY (Section 2.5).

2.1 Encoding of problems

We define a *planning domain* by a set of object types and a library of operators that act on such objects. The PRODIGY language for describing operators is based on the STRIPS domain language [Fikes and Nilsson, 1971], extended to express conditional effects, disjunctive preconditions, and quantifications.

An operator is defined by its *preconditions* and *effects*. The preconditions of an operator are the conditions that must be satisfied before the execution. They are represented by a logical expression with negations, conjunctions, disjunctions, universal quantifiers, and existential quantifiers. The effects are encoded as a list of predicates added to the current state or deleted from the state upon the execution.

We may specify conditional effects, also called *if-effects,* whose outcome depends on the state. An if-effect is defined by its *conditions* and *actions*. If the conditions hold, the effect changes the state according to its actions; else, it does not affect the state.

The effect conditions are represented by a logical expression in the same way as operator preconditions; however, their meaning is somewhat different. If the preconditions of an operator do not hold in the state, the operator cannot be executed. On the other hand, if the conditions of an if-effect do not hold, we can execute the operator, but the if-effect does not change the state. The actions of an if-effect are predicates, to be added to the state or deleted from the state; that is, their encoding is identical to unconditional effects.

In Figure 1, we give an example of a simple domain; its syntax differs slightly from the PRODIGY language [Carbonell *et al.*, 1992] for the purpose of better readability. The domain includes two types of objects, Package and Place, and the Place type has two subtypes, Town and Village. We use types to limit the allowed values of variables in the operator description.

A truck carries packages between towns and villages. The truck's fuel tank is sufficient for only one ride. Towns have gas stations, so the truck can refuel before leaving a town. On the other hand, villages do not have gas stations; if the truck comes to a village without a supply of extra fuel, it cannot leave. To avoid this problem, the truck can get extra fuel in any town. If a package is fragile, it always gets broken during loading. We can cushion a package with soft material, which removes the fragility and prevents breakage.

A *planning problem* is defined by a list of objects, initial state, and goal statement. The *initial state* is a set of literals; the *goal statement* is a condition that must hold after executing a plan. A *solution* is a sequence of instantiated operators that can be executed from the initial state to achieve the goal. We give an example problem in Figure 2; the task is to deliver two packages from town-1 to ville-1. We may solve it as follows: "**load**(pack-1,town-1), **load**(pack-2,town-1), **leave-town**(town-1,ville-1), **unload**(pack-1,ville-1), **unload**(pack-2,ville-1)."

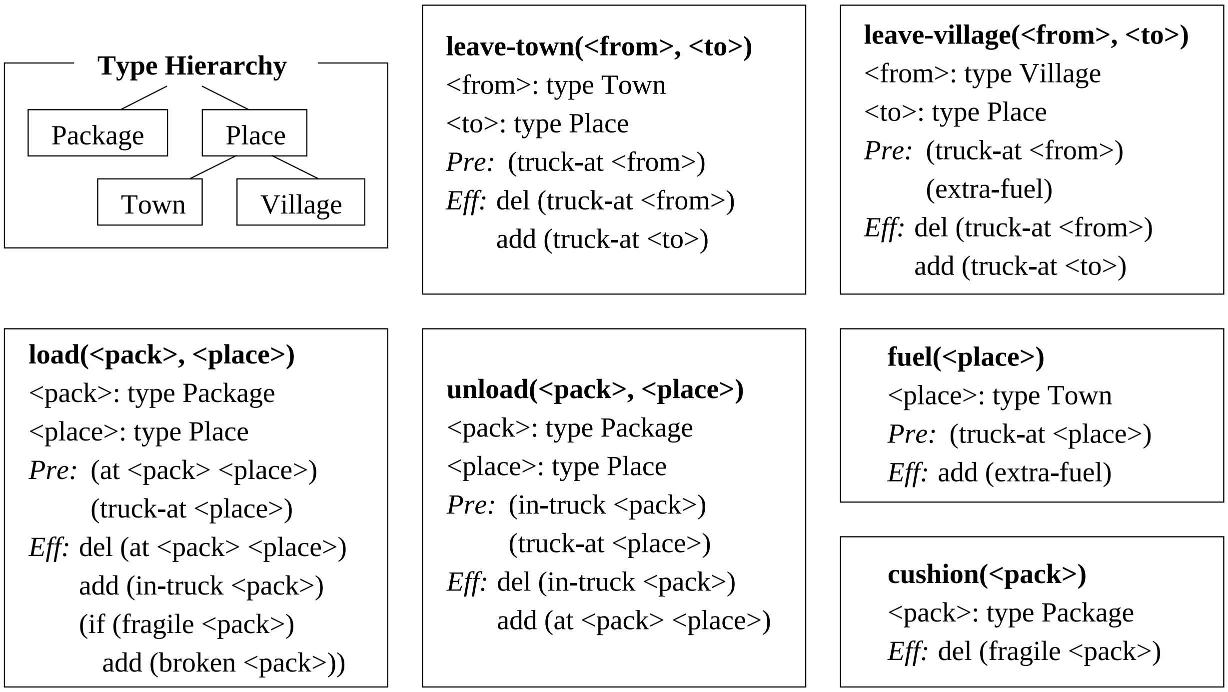


Figure 1: Encoding of a simple trucking world in the PRODIGY domain language. The Trucking Domain is defined by a hierarchy of object types and a library of six operators.

2.2 Incomplete plans

Given a problem, most planners begin with the empty plan and modify it until a solution is found. Examples of modifications include adding an operator, instantiating or constraining a variable in an operator, and imposing an ordering constraint. The intermediate sets of operators are called *incomplete plans*, and we view them as nodes in the search space. Each modification of a current incomplete plan gives rise to a new node, and the number of possible modifications determines the branching factor of search.

Researchers have explored a variety of structures for representing an incomplete plan. For example, it may be a sequence of operators [Fikes and Nilsson, 1971] or a partially ordered set [Tate, 1977]. Some planners fully instantiate operators, whereas others use the unification of operator effects with goals [Chapman, 1987]. Some systems mark relations among operators by causal links [McAllester and Rosenblitt, 1991], and others do not explicitly maintain these relations.

The PRODIGY system also solves a problem by adding operators to the initially empty plan. In PRODIGY, a plan consists of two parts: a total-order *head-plan* and tree-structured *tail-plan* (see Figure 3). The root of the tail-plan's tree is the goal statement G, the other nodes are fully instantiated operators, and the edges are ordering constraints.

The tail-plan is built by a backward chainer, which starts from the goal statement and adds operators, one by one, to achieve goal literals and preconditions of previously added operators. When the algorithm adds an operator to the tail-plan, it *instantiates* the operator, that is, it replaces all variables with specific objects. The preconditions of a fully instantiated operator are a conjunction of literals, where every literal is an instantiated predicate.

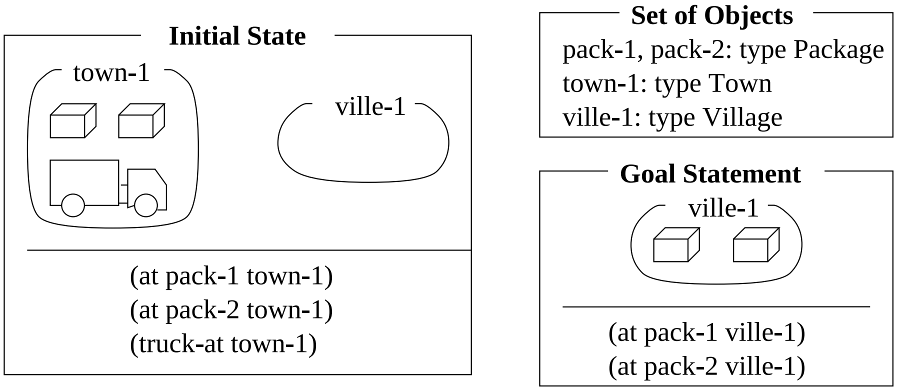


Figure 2: Encoding of a problem in the Trucking Domain, which includes a set of objects, an initial world state, and a goal statement. The task is to deliver two packages from town-1 to ville-1.

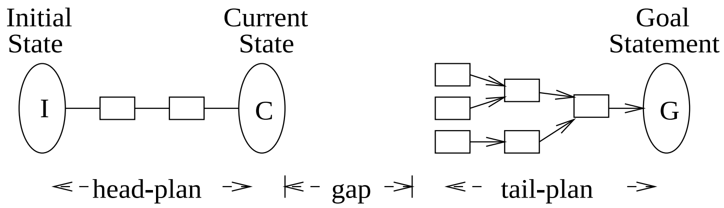


Figure 3: Representation of an incomplete plan. It consists of a total-order head-plan, which can be executed from the initial state, and a tree-structured tail-plan constructed by a backward chainer. The current state C is the result of applying the head-plan operators to the initial state I.

The head-plan is a sequence of instantiated operators that can be executed from the initial state. It is generated by an execution-simulating algorithm, described in Section 2.3. The simulated execution of the head-plan transforms the initial state I into a new state C, called the *current state*. In Figure 4, we show an incomplete plan for the example problem.

Since the head-plan is a total-order sequence of operators that do not contain variables, the current state C is uniquely defined. If the tail-plan operators cannot be executed from the current state C, then there is a "gap" between the head-plan and tail-plan, and the purpose of planning is to bridge this gap. For example, we can bridge the gap in Figure 4 by a sequence of two operators: "**load**(pack-2,town-1), **leave-town**(town-1,ville-1)."

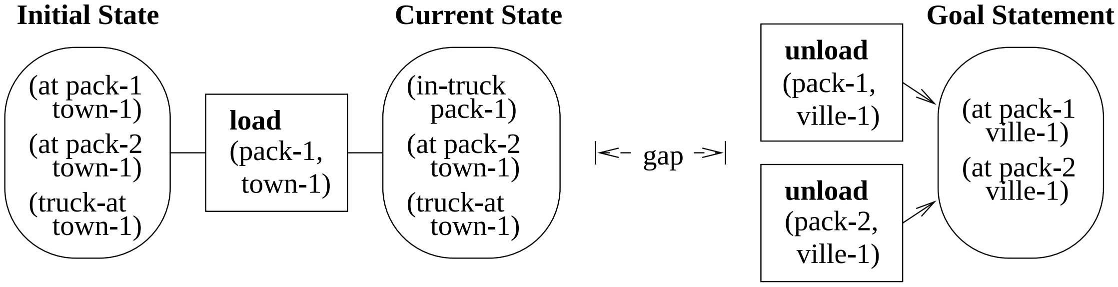


Figure 4: Example of an incomplete plan in the Trucking Domain. The head-plan consists of a single operator, **load**; the tail-plan comprises two **unload** operators, linked to the goal literals.

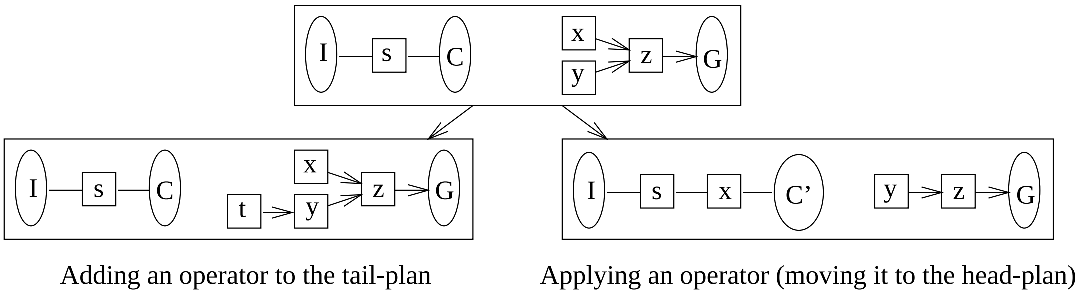


Figure 5: Modifying an incomplete plan. PRODIGY either adds a new operator to the tail-plan tree (left), or moves one of the previously added operators to the head-plan (right).

2.3 Simulating execution

Given an initial state I and a goal statement G, PRODIGY begins with the empty plan and modifies it until it finds a solution. The initial incomplete plan has no operators, and its current state is the same as the initial state, $C = I$.

At each step, PRODIGY can modify the current plan in one of two ways (see Figure 5). First, it can add an operator to the tail-plan (operator t in Figure 5) to achieve a goal literal or a precondition of another operator. This tail-plan modification is a job of the backward-chaining algorithm, described in Section 2.4. Second, PRODIGY can move some operator from the tail-plan to the head-plan (operator x in Figure 5), if its preconditions are satisfied in the current state C. This operator becomes the last in the head-plan, and the current state is updated to account for its effects.

Intuitively, we may imagine that the system executes the head-plan operators in the real world, and it has already changed the world from the initial state I to the current state C. If the tail-plan contains an operator whose preconditions are satisfied in C, then PRODIGY applies this operator and further changes the state. Because of this analogy with real-world changes, moving an operator to the head-plan is called the *application* of the operator; however, this term refers to *simulating* the application. Even if the execution of the head-plan is disastrous, the world does not suffer; the planner can backtrack and try an alternative execution sequence.

When the system applies an operator to the current state, it begins with the deletion effects and removes the corresponding literals from the state. Then, it performs the addition of new literals. Thus, if the operator adds and deletes the same literal, the net result is adding it to the state. For example, suppose that the current state includes the literal (truck-at town-1), and PRODIGY applies **leave-town**(town-1,town-1), whose effects are "del (truck-at town-1)" and "add (truck-at town-1)." The planner first removes this literal from the state description, and then adds it back. If PRODIGY processed the effects in the opposite order, it would permanently remove the truck's location, thus getting an inconsistent state.

An operator application is the only way of updating the head-plan. The planner never inserts a new operator directly into the head-plan, which means that it uses only goal-relevant operators in the forward chaining. The search terminates when the head-plan achieves the goals, that is, the goal statement G is satisfied in C. If the tail-plan is not empty at that point, it is dropped.

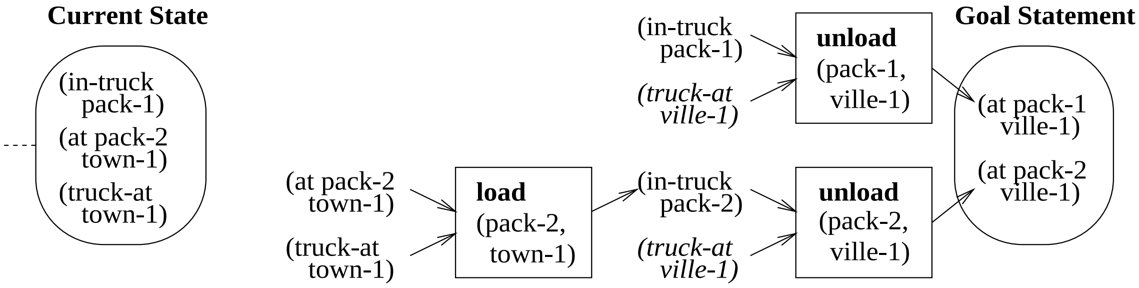


Figure 6: Example of a tail-plan for a trucking problem. First, the backward chainer adds the two **unload** operators to achieve the goal literals. Then, it inserts **load** to achieve the precondition (in-truck pack-2) of **unload**(pack-2,ville-1).

2.4 Backward chaining

We next describe the backward-chaining procedure that constructs a tree-structured tail-plan. When the planner invokes this procedure, it adds a new operator to the tail-plan for achieving either a goal literal or a precondition of another tail-plan operator. Then, it establishes a link from the newly added operator to the achieved literal and adds the corresponding ordering constraint. For example, if the plan is as shown in Figure 4, the procedure may add **load**(pack-2,town-1) to achieve the precondition (in-truck pack-2) of **unload**(pack-2,ville-1) (see Figure 6). If the planner uses an if-effect to achieve a literal, then the effect's conditions are added to the preconditions of the instantiated operator.

PRODIGY tries to achieve a literal only if (1) it is not true in the current state C and (2) it has not been linked to any tail-plan operator. Unsatisfied goal literals and preconditions are called *subgoals*. For example, the tail-plan in Figure 6 has two identical subgoals, marked by italics.

Before inserting an operator into the tail-plan, PRODIGY fully instantiates it, that is, substitutes all variables with specific objects. Since PRODIGY allows disjunctive and quantified preconditions, instantiating an operator may be a difficult problem. The system uses a constraint-based matching procedure that generates all possible instantiations [Wang, 1992].

For example, suppose that the backward chainer uses an operator **load**(<pack>,<place>) to achieve the subgoal (in-truck pack-2) (see Figure 7). PRODIGY instantiates the variable <pack> with the object pack-2 from the subgoal literal, and then it has to instantiate the other variable, <place>. Since the domain has two places, town-1 and ville-1, the variable has two possible instantiations, which give rise to different branches in the search space (Figure 7).

In Figure 8, we summarize the search algorithm that explores the space of possible plans. The *Operator-Application* procedure builds the head-plan and maintains the current state, whereas *Backward-Chainer* constructs the tail-plan.

The algorithm includes five decision points, which give rise to different branches of the search space. It can backtrack over the decision to apply an operator (Line 2a) and over the choice of an "applicable" tail-plan operator (Line 1b). It also backtracks over the choice of a subgoal (Line 1c), an operator that achieves it (Line 2c), and the operator's instantiation (Line 4c). We summarize the decision points in Figure 9.

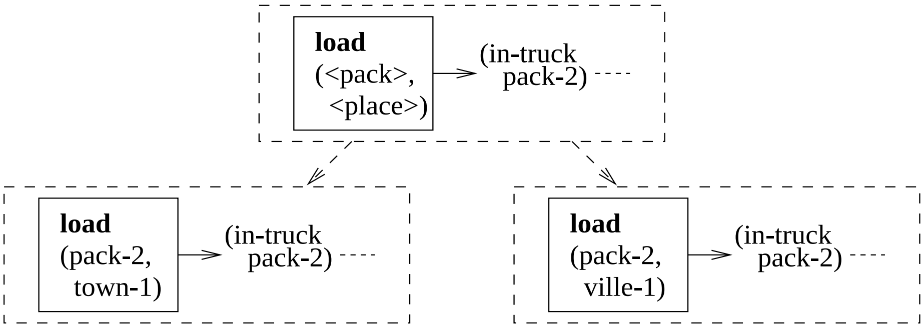


Figure 7: Instantiating a newly added operator. If the set of objects is as shown in Figure 2, PRODIGY can generate two alternative versions of **load** for achieving the subgoal (in-truck pack-2).

The first two choices (Lines 2a and 1b) enable the planner to consider different orderings of operators in the head-plan. These choices are essential for planning with interacting subgoals; they are analogous to the choice of ordering constraints in least-commitment planners.

2.5 Main versions

The algorithm in Figure 8 has five decision points, which allow flexible selection of operators, their instantiations, and the order of their execution; however, these decisions give rise to a large branching factor. The use of built-in heuristics, which eliminate some choices, may reduce the search space and improve the efficiency. On the negative side, such heuristics prune some solutions, and they may direct the search to a suboptimal solution or even prevent finding any solution. Setting appropriate restrictions on the planner's choices is a major research problem.

Although the described algorithm underlies all PRODIGY versions, from PRODIGY1 to FLECS, the versions differ in their use of decision points and heuristics. Researchers investigated different trade-offs between flexibility and reduction of branching. They gradually increased the number of available decision points from two in PRODIGY1 to all five in FLECS.

The versions also differ in some features of the domain language, in the use of learning modules, and in the low-level implementation of search mechanisms. We do not discuss these differences; the reader may learn about them from the review article by Veloso *et al.* [1995].

2.5.1 PRODIGY1 and PRODIGY2

The early versions of PRODIGY had only two backtracking points: the choice of an operator (Line 2c in Figure 8) and the instantiation of the selected operator (Line 4c). The other three decisions were based on fixed heuristics, which did not give rise to multiple search branches. The algorithm preferred operator application to adding new operators (Line 2a), applied the tail-plan operator that had been added last (Line 1b), and achieved the first unsatisfied precondition of the last added operator (Line 1c). This algorithm generated suboptimal solutions and sometimes failed to find any solution.

For example, consider the PRODIGY2 search for the problem in Figure 2. The planner adds **unload**(pack-1,ville-1) to achieve (at pack-1 ville-1), and **load**(pack-1,town-1) to achieve

Base-PRODIGY
1a. If the goal statement G is satisfied in the current state C, then return the head-plan.
2a. Either
    (i) *Backward-Chainer* adds an operator to the tail-plan, or
    (ii) *Operator-Application* moves an operator from the tail-plan to the head-plan.
  Decision point: Choose between (i) and (ii).
3a. Recursively call *Base*-PRODIGY on the resulting plan.

Operator-Application
1b. Pick an operator *op* in the tail-plan, such that
    (i) there is no operator in the tail-plan ordered before *op*, and
    (ii) the preconditions of *op* are satisfied in the current state C.
  Decision point: Choose one of such operators.
2b. Move *op* to the end of the head-plan and update the current state C.

Backward-Chainer
1c. Pick a literal l among the current subgoals.
  Decision point: Choose one of the subgoal literals.
2c. Pick an operator *op* that achieves l.
  Decision point: Choose one of such operators.
3c. Add *op* to the tail-plan and establish a link from *op* to l.
4c. Instantiate the free variables of *op*.
  Decision point: Choose an instantiation.
5c. If the effect that achieves l has conditions,
    then add them to the operator preconditions.

Figure 8: Foundations of the PRODIGY planner. The *Operator-Application* procedure simulates execution of operators, whereas *Backward-Chainer* selects operators relevant to the goal.

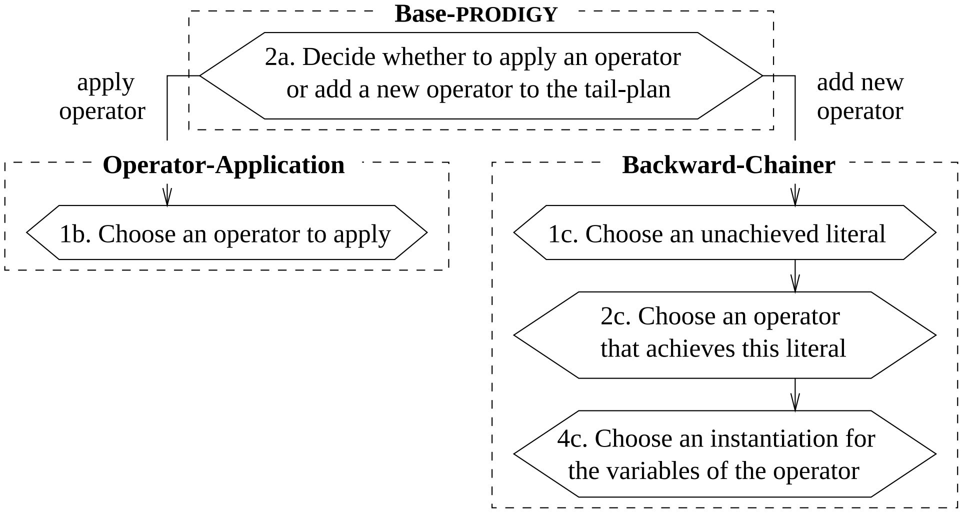


Figure 9: Decision points of the PRODIGY algorithm, given in Figure 8. Every decision point allows backtracking, thus giving rise to multiple branches of the search space.

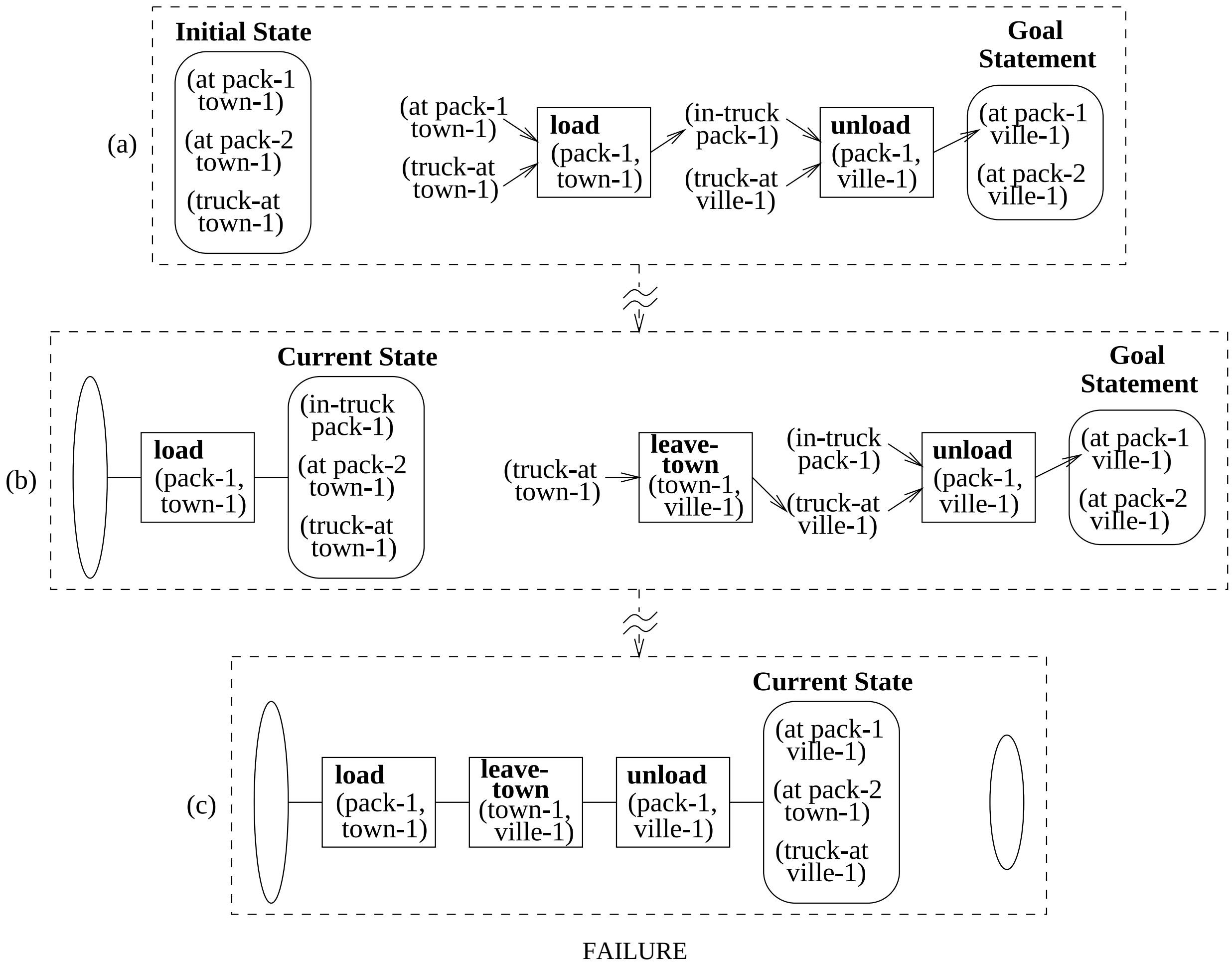


Figure 10: Incompleteness of PRODIGY2. The planner fails to solve the trucking problem in Figure 2. Since PRODIGY2 always prefers the operator application to adding new operators, it cannot load both packages before driving the truck to its destination.

the precondition (in-truck pack-1) of **unload** (see Figure 10a). Then, it applies **load** and adds **leave-town**(town-1,ville-1) to achieve the precondition (truck-at ville-1) of **unload** (Figure 10b). Finally, PRODIGY applies **leave-town** and **unload** (Figure 10c), thus bringing only one package to the village.

Since the algorithm uses only two backtracking points, it does not consider loading two packages before the ride or getting extra fuel before leaving the town; thus, it fails to solve the problem. This example demonstrates that the PRODIGY2 system is *incomplete*; that is, it may fail on a problem that has a solution. The user may improve the situation by providing domain-specific control rules that enforce different choices of subgoals in Line 1c. Note that PRODIGY2 does not backtrack over these choices, and an inappropriate control rule may cause a failure. This approach often improves the performance; however, it requires the user to assume the responsibility for completeness and plan quality.

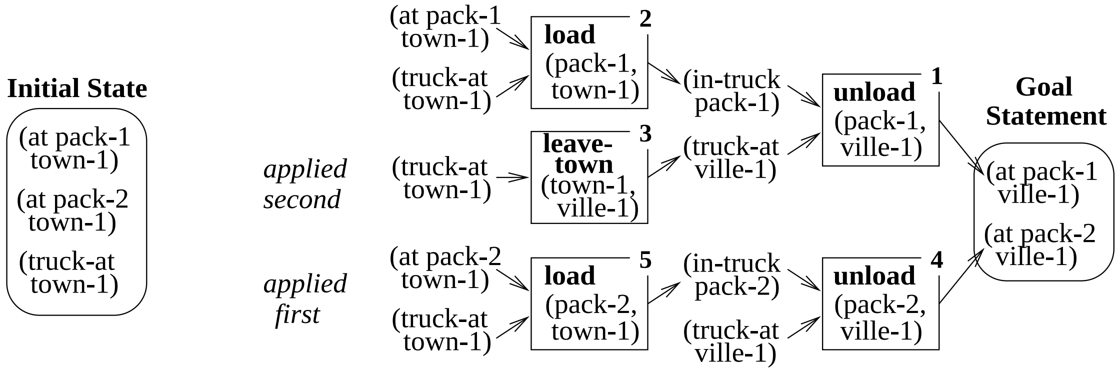


Figure 11: Example of inefficiency in PRODIGY4. The boldface numbers, in the upper right corners of operators, mark the order of adding operators to the tail-plan. Since PRODIGY4 always applies the last added operator, it tries to apply **leave-town** before one of the **load** operators, which leads to a deadend and requires backtracking.

2.5.2 NOLIMIT and PRODIGY4

During the work on NOLIMIT, Veloso added two more backtracking points: delaying the application of tail-plan operators (Line 2a) and choosing a subgoal (Line 1c); later, PRODIGY4 inherited these points. On the other hand, PRODIGY4 makes no decision in Line 1b; it always applies the last added operator. The absence of this decision point does not rule out any solutions, but it may negatively affect the search time.

For instance, if we use PRODIGY4 to solve the problem in Figure 2, it may generate the tail-plan in Figure 11, where the numbers show the order of adding operators. We could now solve the problem by applying the two **load** operators, the **leave-town** operator, and then both **unload** operators; however, the planner cannot use this order. It applies **leave-town** before one of the **load** operators, which leads to a deadend. It then has to backtrack and construct a new tail-plan that allows the right order of applying the operators.

2.5.3 FLECS

The FLECS planner has all five decision points, but it does not backtrack over the choice of a subgoal (Line 1c), which means that only four points give rise to multiple search branches. Since backtracking over these points may produce an impractically large space, Veloso and Stone [1995] implemented general heuristics that further limit the space.

They experimented with two versions of the FLECS planner, SAVTA and SABA, which differ in their choice between adding an operator to the tail-plan and applying an operator (Line 2a). SAVTA prefers to apply tail-plan operators before adding new ones, whereas SABA tries to delay their application.

Experiments have shown that the greater flexibility of PRODIGY4 and FLECS usually has an advantage over PRODIGY2, despite the larger branching factor. The relative effectiveness of PRODIGY4, SAVTA, and SABA depends on a specific domain, and the choice among them is often essential for performance [Stone *et al.*, 1994].

cushion(<pack>, <place>)
<pack>: type Package
<place>: type Place
Pre: (truck-at <place>)
(or (at <pack> <place>)
(in-truck <pack>))
Eff: del (fragile <pack>)

(a) Disjunction.

Goal Statement
(exists <pack> of type Package
(at <pack> ville-1))

(b) Existential quantification.

Goal Statement
(forall <pack> of type Package
(at <pack> ville-1))

(c) Universal quantification.

Figure 12: Examples of disjunction and quantification in PRODIGY. The user may utilize them in precondition expressions and goal statements.

3 Extended domain language

The PRODIGY domain language is an extension of the STRIPS language [Fikes and Nilsson, 1971]. The STRIPS planner used a limited description of operators, and PRODIGY researchers added several advanced features, including complex preconditions and goal expressions (Section 3.1), inference rules (Section 3.2), and flexible use of object types (Section 3.3).

3.1 Extended operators

The PRODIGY language allows complex logical expressions in operator preconditions, if-effect conditions, and goal statements. They may include not only negations and conjunctions, but also disjunctions and quantifications. The language also enables the user to specify the costs of operators, which serve as a measure of plan quality.

3.1.1 Disjunctive preconditions

To illustrate disjunction, we consider a variation of the **cushion** operator, given in Figure 12(a). In this example, we can cushion a package when it is inside or near the truck.

When PRODIGY instantiates an operator that has disjunctive preconditions, it generates an instantiation for one element of the disjunction and discards all other elements. For example, if the planner has to cushion pack-1, it may choose the instantiation (at pack-1 town-1), which matches (at <pack> <place>), and discard the other element, (in-truck <pack>).

If the initial choice does not lead to a solution, the planner backtracks and considers the instantiation of another element. For instance, if the selected version of **cushion** has proved inadequate, PRODIGY may discard the first element of the disjunction, (at <pack> <place>), and choose the instantiation (in-truck pack-1) of the other element.

3.1.2 Quantified preconditions

We illustrate the use of quantifiers in Figures 12(b) and 12(c). In the first example, the planner has to transport *any* package to ville-1; in the second, it has to deliver *all* packages.

When the planner instantiates an existential quantifier, it selects one object of the specified type. For example, it may decide to deliver pack-1 to ville-1, thus replacing the goal in Figure 12(b) by (at pack-1 ville-1). If it does not lead to a solution, PRODIGY backtracks and tries another object. When instantiating a universally quantified expression, the planner treats it as a conjunction over all matching objects.

3.1.3 Instantiated operators

The PRODIGY language allows arbitrary logical expressions, which may contain multiple levels of negations, conjunctions, disjunctions, and quantifications. When adding an operator to the tail-plan, PRODIGY generates all possible instantiations of its preconditions and chooses one of them. If the planner backtracks, it chooses an alternative instantiation. Every instantiation is a conjunction of literals, some of which may be negated; it has no disjunctions, quantifications, or negated conjunctions.

Wang [1992] designed an advanced algorithm for generating possible instantiations of operators and goal statements. In particular, she developed a mechanism for pruning inconsistent choices of objects, and heuristics for selecting the most promising instantiations.

3.1.4 Costs

The use of costs allows measuring the plan quality. We assign nonnegative numeric costs to operators and define a plan cost as the sum of its operator costs. The lower the cost, the better the plan.

The authors of the original PRODIGY architecture did not provide support for costs, and they usually measured the plan quality by the number of operators. Pérez [1995] implemented costs during her exploration of control knowledge for improving the plan quality; however, she did not incorporate costs into the main version. Fink [2003] re-implemented the cost mechanism during his work on automatic representation changes.

In Figure 13, we give an example of cost encoding. For every operator, the user specifies a Lisp function, whose arguments are operator variables. Given specific objects, the function returns the corresponding cost, which must be a nonnegative real number. If the cost is the same for all instantiations of an operator, it may be specified by a number rather than a function.

The example in Figure 13(a) includes two cost functions, called *leave-cost* and *load-cost*. We give pseudocode for these functions in Figure 13(b), and their actual Lisp encoding in Figure 13(c). The cost of driving between two locations is linear in the distance, determined by the *miles* function. The user may specify distances by a matrix or by a list of initial-state literals, and she should provide the appropriate look-up procedure. The loading cost depends on the location type; it is larger in villages. Finally, the cost of the **cushion** operator is constant.

When the planner instantiates an operator, it calls the corresponding function to determine the cost of the resulting instantiation. If the returned value is negative, the system signals an error. Since plans consist of fully instantiated operators, the planner can determine the cost of every intermediate plan.

leave-town(<from>, <to>)
<from>: type Town
<to>: type Place
...
Cost: leave-cost(<from>,<to>)

load(<pack>, <place>)
<pack>: type Package
<place>: type Place
...
Cost: load-cost(<place>)

cushion(<pack>)
<pack>: type Package
...
Cost: 5

(a) Use of cost functions and constant costs.

leave-cost *(<from>,<to>)*
Return 0.2 · *miles(<from>,<to>)* + 5.

load-cost*(<place>)*
If *<place>* is of type Village,
then return 4; else, return 3.

(b) Pseudocode for the cost functions.

```
(defun leave-cost (<from> <to>)
  (+ (* 0.2 (miles <from> <to>)) 5))

(defun load-cost (<place>)
  (if (eq (type-name (object-type <place>)) 'Village)
    4 3))
```

(c) Actual Lisp functions.

Figure 13: Encoding of operator costs. The user may specify a constant cost or, alternatively, a Lisp function that inputs operator variables and returns a nonnegative cost value. If an operator description does not include a cost, PRODIGY assumes that it is 1.

3.2 Inference rules

The PRODIGY language supports two mechanisms for changing the current state: operators and inference rules. They have identical syntax but differ in semantics. Operators encode actions that change the world, whereas rules point out implicit properties of the world.

3.2.1 Example

In Figure 14, we show three inference rules for the Trucking Domain. In this example, we have made two modifications to the original domain (see Figure 1). First, the **cushion** operator adds (cushioned <pack>) instead of deleting (fragile <pack>), and the **add-fragile** rule indicates that uncushioned packages are fragile. Second, the domain includes the type County and the predicate (within <place> <county>). We use the **add-truck-in** rule to infer the county of the truck's current location. For example, if the truck is at town-1, and town-1 is within county-1, then the rule adds (truck-in county-1). Similarly, we use **add-in** to infer the current county of each package.

3.2.2 Use of inferences

The encoding of inference rules is the same as that of operators, which may include disjunctions, quantifiers, and if-effects; however, the rules have no costs, and they do not affect the overall plan cost.

The use of inference rules is also similar to that of operators: the planner adds an instantiated rule to the tail-plan, for achieving the selected subgoal, and applies the rule when its preconditions hold in the current state. We illustrate it in Figure 15, where the

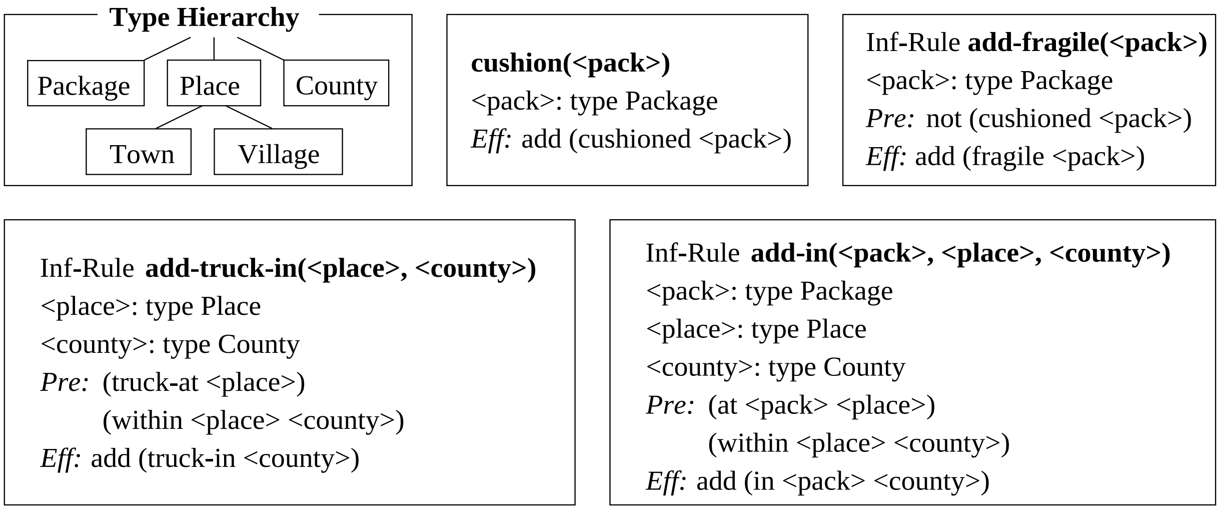


Figure 14: Encoding of inference rules in PRODIGY. These rules point out indirect results of changing the world state; their syntax is identical to that of operators.

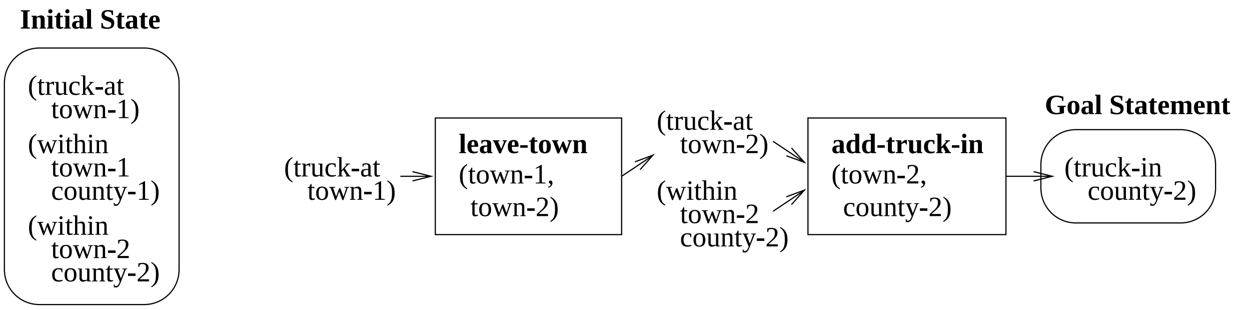


Figure 15: Use of an inference rule in backward chaining. PRODIGY links the **add-truck-in** rule to the goal literal and then adds **leave-town** to achieve the rule's precondition (truck-at town-2).

planner uses the **add-truck-in** rule to achieve the goal, and then adds **leave-town** to achieve the rule's precondition.

If the system applies an inference rule and *later* adds an operator that invalidates the rule's preconditions, then it removes the rule's effects from the state. For example, the inference rule in Figure 16(a) adds (truck-in town-2) to the state. If the system then applies **leave-town** (Figure 16b), it negates the preconditions of **add-truck-in** and cancels its effects. This semantics differs from operators, whose effects remain in the state, unless deleted by opposite effects of other operators.

3.2.3 Eager and lazy rules

The *Backward-Chainer* algorithm selects rules at its discretion and may disregard unwanted rules. On the other hand, if some rule has an undesirable effect, it should be applied regardless of the planner's choice. For example, if pack-1 is not cushioned, the system should immediately add (fragile pack-1).

When the user encodes a domain, she has to mark all rules that have unwanted effects.

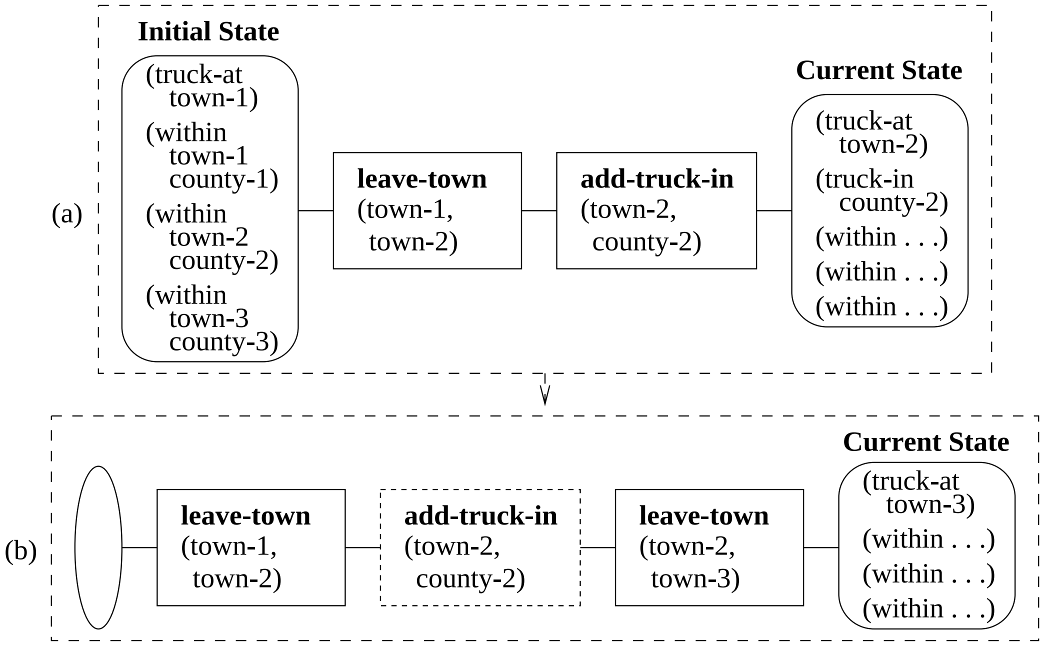


Figure 16: Cancelling the effects of an inference rule upon the negation of its preconditions. When PRODIGY applies **leave-town**(town-2,town-3), it negates the precondition (truck-at town-2) of the **add-truck-in** rule; hence, it removes the effects of this rule from the current state.

When the preconditions of a marked rule hold in the current state, the system applies it at once, even if it is not in the tail-plan. The marked rules are called *eager*, whereas the other rules are *lazy*. Note that *Backward-Chainer* may use both eager and lazy rules, and the only special property of eager rules is their forced application in the matching states.

3.2.4 Truth maintenance

When PRODIGY applies an operator or inference rule, it updates the current state and then identifies the previously applied rules whose preconditions no longer hold. If the system finds such rules, it modifies the state by removing their effects. Next, PRODIGY looks for an eager rule whose conditions hold in the resulting state. If the system finds such a rule, it applies the rule and further changes the state.

If inference rules interact with each other, this process may involve a chain of rule applications and cancellations. It terminates when the system gets to a state that does not require applying a new eager rule or removing the effects of old rules. This chain of state modifications does not involve search; it is similar to the firing of productions in the SOAR system [Laird *et al.*, 1986; Golding *et al.*, 1987]. Blythe designed a truth-maintenance procedure that keeps track of all applicable rules and controls this forward chaining.

If the user provides inference rules, she has to ensure that the inferences are consistent. In particular, a rule must not negate its own preconditions. If two rules may be applied in the same state, they must not have opposite effects. If a domain includes several eager rules, they should not cause an infinite cyclic chain of forced applications. PRODIGY does not check for such inconsistencies, and inappropriate rules may cause unpredictable results.

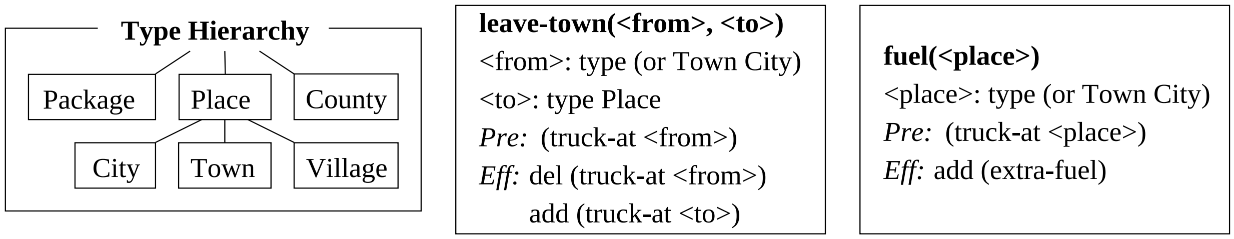


Figure 17: Disjunctive type. The <from> and <place> variables are declared as "(or Town City)," which means that they may be instantiated with objects of two simple types, Town and City.

3.3 Complex types

We have already described a type hierarchy (see Figure 1), which defines object classes and enables the user to limit the allowed values of variables. For example, the possible values of the <from> variable in **leave-town** include all towns, but not villages. The early versions of PRODIGY did not support a type hierarchy. Veloso designed a typed domain language during her work on NOLIMIT, and the authors of PRODIGY4 further developed this language.

A type hierarchy is a tree, whose nodes are called *simple types*. For instance, the hierarchy in Figure 1 has five simple types: Package, Town, Village, Place, and the root type that includes all objects. We have illustrated the use of simple types; however, they often do not provide sufficient flexibility. For example, consider the hierarchy in Figure 17 and suppose that the truck may get extra fuel in a town or city, but not in a village. We cannot encode this constraint with simple types, unless we define an additional type. The PRODIGY language includes a mechanism for specifying complex constraints through disjunctive and functional types.

3.3.1 Disjunctive types

In Figure 17, we illustrate a disjunctive type that determines the possible values of the variables <from> and <place>. The user specifies a disjunctive type as a set of simple types; in the example, it includes Town and City. When the planner instantiates the corresponding variable, it uses an object that belongs to any of these types. For instance, the planner may use **leave-town** for departing from a town or city.

3.3.2 Functional types

In Figure 18, we give an example of a functional type that limits the values of the <to> variable. The description of a functional type consists of two parts: a simple or disjunctive type and a boolean test function. The system first identifies all objects of the specified simple or disjunctive type, and then eliminates the objects that do not satisfy the test function. The remaining objects are the valid values of the declared variable. In the example, the valid values of <to> include all places that have road connections with <from>. For instance, if every place is connected with every other place except itself, then we use the test function given in Figure 18(b). The domain encoding must include a Lisp implementation of this function, as shown in Figure 18(c).

```
leave-town(<from>, <to>)
<from>: type (or Town City)
<to>: type Place
        connected(<from>,<to>)
Pre:  (truck-at <from>)
Eff:  del (truck-at <from>)
      add (truck-at <to>)
```

(a) Use of a functional type.

```
connected(<from>, <to>)
If  <from> = <to>,
   then return False;
   else, return True.
```

(b) Pseudocode for the function.

```
(defun connected (<from> <to>)
   (not (eq <from> <to>)))
```

(c) Actual Lisp function.

Figure 18: Functional type. When PRODIGY instantiates **leave-town**, it ensures that <from> and <to> are connected by a road. The user has to implement a boolean Lisp function for testing the connectivity. We give an example function that defines the fully connected graph of roads.

The boolean function is an arbitrary Lisp procedure whose arguments are operator variables. The function *must* input the variable described by the functional type. In addition, it may input variables declared before this functional type; however, the function cannot input variables declared after it. For example, we use the <from> variable in limiting the values of <to>; however, we cannot use the <to> variable as an input to a test function for <from> because of the declaration order.

3.3.3 Use of test functions

When the planner instantiates a variable with a functional type, it identifies all objects of the specified simple or disjunctive type, prunes the objects that do not satisfy the test function, and then selects an object from the remaining set. If the user specifies not only functional types but also control rules, which further limit suitable instantiations, then the generation of instantiated operators becomes a complex matching problem. Wang [1992] investigated it and developed an efficient matching algorithm.

Test functions may use any information about the current incomplete plan, other nodes in the search space, and the global state of the system, which allows unlimited flexibility in constraining operator instantiations. In particular, they enable the user to encode functional effects, that is, operator effects that depend on the current state.

3.3.4 Generator functions

The system also supports *generator functions* in the specification of variable types. These functions generate and return a set of allowed values, instead of testing the available values. The user has to specify a simple or disjunctive type along with a generator function. When the system uses the function, it checks whether all returned objects belong to the specified type and prunes the extraneous objects.

In Figure 19, we give an example that involves both a test function, called *positive,* and a generator function, called *decrement.* In this example, the system keeps track of the available space in the trunk. If there is no space, it cannot load packages. We use the

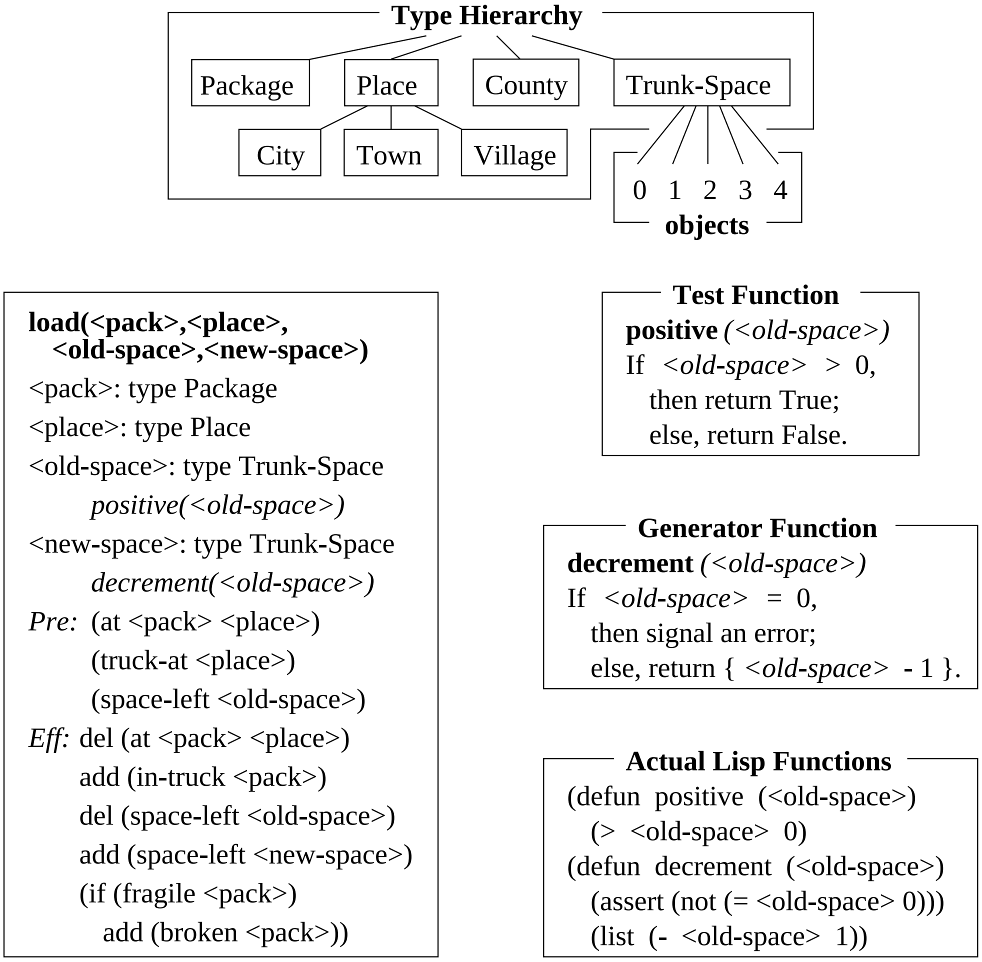


Figure 19: Generator function. The user provides a Lisp function, called *decrement*, that generates instantiations of <new-space>; they must belong to the specified type, Trunk-Space.

generator function to decrement the available space after loading a package. The function always returns one value, which represents the remaining space.

When the user specifies a simple or disjunctive type for a generator function, she may define a numeric type that includes infinitely many values. For instance, the Trunk-Space type in Figure 19 may comprise all natural numbers. On the other hand, the generator function must always return a finite set. The PRODIGY manual [Carbonell *et al.*, 1992] contains a more detailed description of infinite types.

4 Search control

The planning efficiency depends on the search space and the order of expanding nodes of the space. The nondeterministic PRODIGY algorithm in Figure 8 defines the search space, but it does not specify the exploration order. It has several decision points (see Figure 9), which require heuristics for selecting appropriate branches of the space.

The PRODIGY architecture includes a variety of search-control mechanisms, which com-

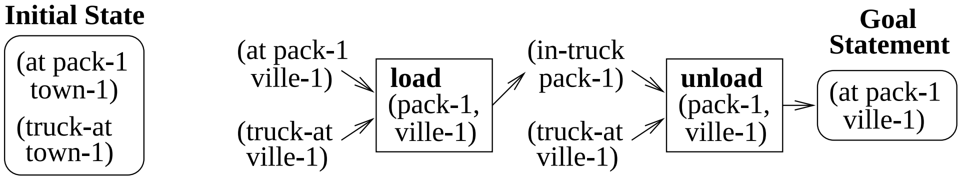


Figure 20: Goal loop in the tail-plan. The precondition (at pack-1 ville-1) of **load** is the same as the goal literal; hence, the planner has to backtrack and choose another operator.

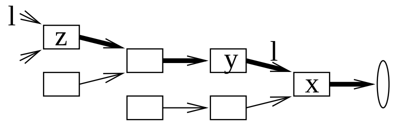


Figure 21: Detection of goal loops. The backward chainer compares the precondition literals of a newly added operator z with the links between z and the goal statement. If some precondition l is identical to one of the link literals, the planner backtracks.

bine general heuristics, domain-specific experience, and advice by the user. Some basic mechanisms are hard-coded into the system; however, most heuristics are optional, and the user can enable or disable them.

We outline some control techniques, including heuristics for avoiding redundant search (Section 4.1), main knobs for adjusting the search strategy (Section 4.2), and control rules (Section 4.3). The reader may find an overview of other control techniques in the article by Blythe and Veloso [1992], which explains dependency-directed backtracking, limited look-ahead, and heuristics for choosing appropriate subgoals and instantiations.

4.1 Avoiding redundant search

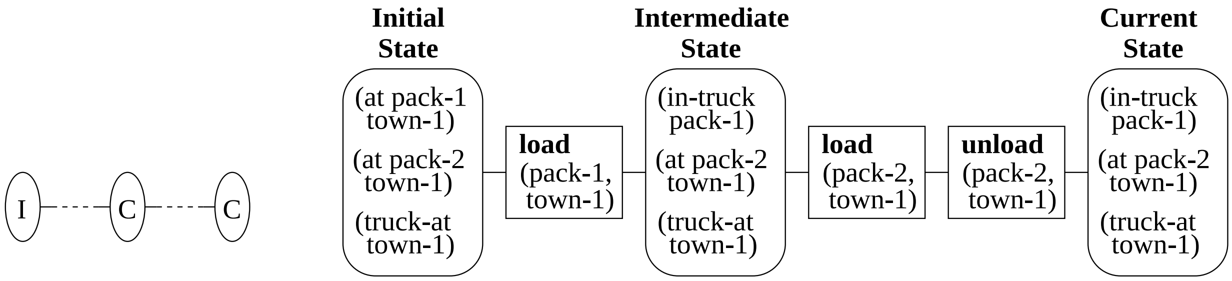
We describe three basic techniques for eliminating redundant branches of the search space, which are hard-coded into the planner.

4.1.1 Goal loops

We first present a mechanism that prevents PRODIGY from running in simple circles. To illustrate it, consider the problem of delivering pack-1 from town-1 to ville-1 (see Figure 20). The planner first adds **unload**(pack-1,ville-1), and then tries to achieve its precondition (in-truck pack-1) by **load**(pack-1,ville-1); however, the precondition (at pack-1 ville-1) of **load** is identical to the goal, and achieving it is as difficult as solving the original problem.

We call it a *goal loop,* which arises when a precondition of a newly added operator is identical to the literal of some link on the path from this operator to the goal statement. We illustrate it in Figure 21, where thick links mark the path from a new operator z to the goal. The precondition l of z makes a loop with an identical precondition of x, achieved by y.

When the planner adds an operator to the tail-plan, it compares the operator's preconditions with the links between this operator and the goal. If the planner detects a goal loop, it backtracks and tries either a different instantiation of the operator or an alternative operator



(a) State loop.

(b) Example of a loop.

Figure 22: State loops in the head-plan. If the current state C is the same as one of the previous states, the planner backtracks. For example, if PRODIGY applies **unload**(pack-2,town-1) immediately after **load**(pack-2,town-1), it creates a state loop.

that achieves the same subgoal. For example, the planner may generate a new instantiation of the **load** operator, **load**(pack-1,town-1).

4.1.2 State loops

The planner also watches for loops in the head-plan, called *state loops*. Specifically, it verifies that the current state differs from all previous states. If the current state is identical to some earlier state (see Figure 22a), the planner discards the current plan and backtracks.

We illustrate a state loop in Figure 22(b), where the application of two opposite operators, **load** and **unload**, leads to a repetition of an intermediate state. The planner would detect this redundancy and either delay the application of **unload** or use a different instantiation.

4.1.3 Satisfied links

Next, we describe the detection of redundant tail-plan operators, illustrated in Figure 23. In this example, PRODIGY is solving the problem in Figure 2, and it has constructed the tail-plan shown in Figure 23(a). The literal (truck-in ville-1) is a precondition of two different operators, **unload**(pack-1,ville-1) and **unload**(pack-2,ville-1). Thus, the plan includes two identical subgoals, and the planner adds two copies of **leave-town**(town-1,ville-1) to achieve these subgoals.

Such situations arise because PRODIGY connects each tail-plan operator to only one subgoal, which simplifies the maintenance of links. When the planner applies an operator, it skips redundant parts of the tail-plan. For example, suppose that it has applied the two **load** operators and one **leave-town**, as shown in Figure 23(b). The precondition (truck-in ville-1) of the tail-plan operator **unload** now holds in the current state, and the planner skips the tail-plan operator **leave-town** linked to this precondition.

When a tail-plan operator achieves a precondition that holds in the current state, we call the corresponding link *satisfied*. We show this situation in Figure 24(a), where the precondition l of x is satisfied, which makes the dashed operators redundant.

The planner keeps track of satisfied links and updates their list after each modification of the current state. When PRODIGY selects a tail-plan operator to apply (line 1b in Figure 8)

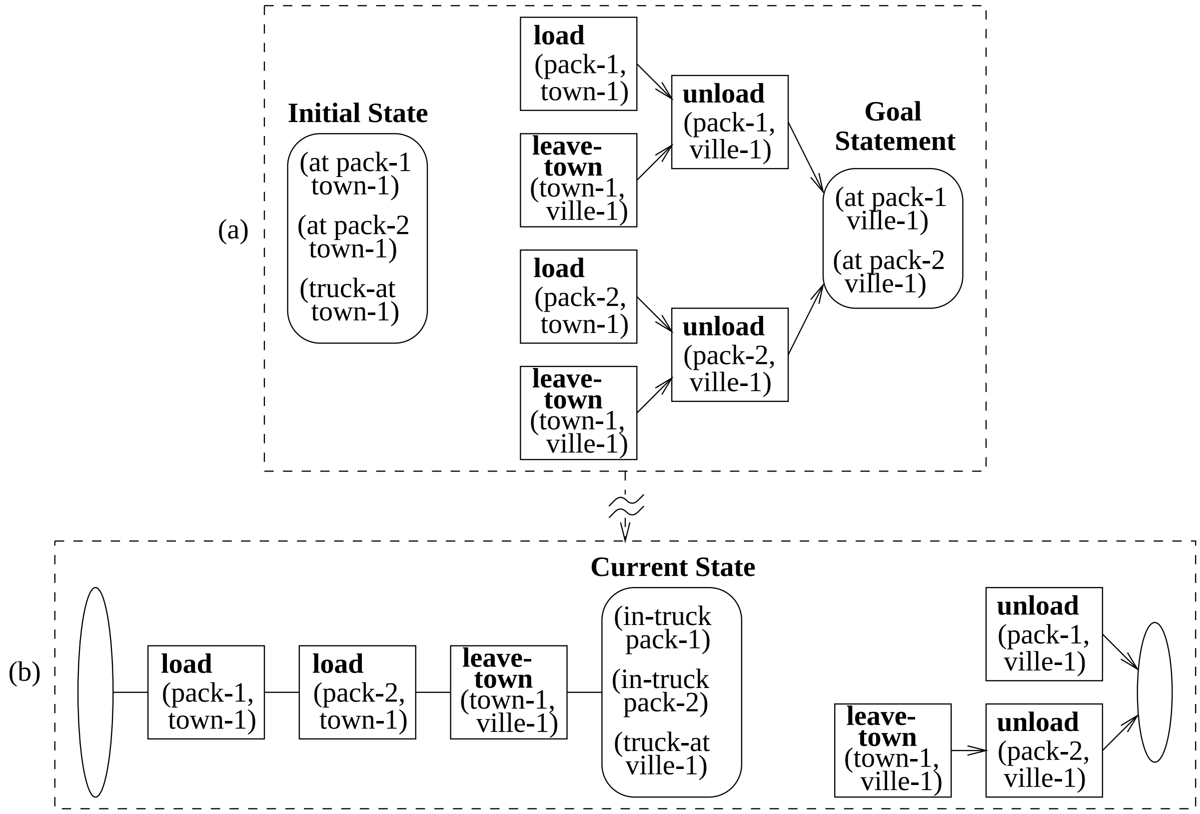


Figure 23: Satisfied link. After the planner has applied three operators, it notices that all preconditions of **unload**(pack-2,ville-1) hold in the current state; hence, it omits the tail-plan operator **leave-town**, which is linked to a satisfied precondition of **unload**.

or a subgoal to achieve (line 1c), it ignores the tail-plan branches that support satisfied links. Thus, it would not consider the dashed operators in Figure 24 and their preconditions.

If the planner applies the operator x, it discards the dashed branch that supports a precondition of x (Figure 24b). Note that the planner discards the dashed branch only after applying x. If it decides to apply some other operator before x, it may delete l, in which case dashed operators become useful again.

4.2 Knob values

The PRODIGY architecture includes several knob variables, which allow the user to adjust the search strategy. Some knobs are numeric values, such as the search depth, whereas others specify the choices among alternative heuristics.

4.2.1 Depth limit

The user usually limits the search depth, which results in backtracking upon reaching the pre-set limit. Note that the number of operators in a solution is proportional to the search depth; hence, limiting the depth is equivalent to limiting the solution length.

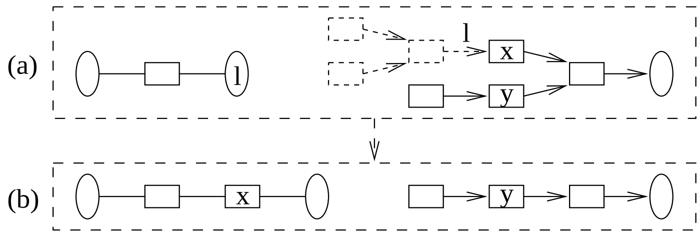


Figure 24: Identification of satisfied links. PRODIGY keeps track of all link literals satisfied in the current state, and disregards the tail-plan operators that support these literals.

After adding operator costs to the PRODIGY language, Fink provided a knob for limiting the plan cost. If the system constructs a plan whose cost is greater than the limit, it backtracks and considers an alternative branch. If the user bounds both search depth and plan cost, the planner backtracks upon reaching either limit.

The effect of these bounds varies across domains. In some domains, they improve efficiency by preventing a long descent into a branch that has no solutions. In other domains, they cause an extensive search instead of a fast generation of a suboptimal plan. If the search space has no solution within the specified bounds, the system fails to solve the problem; thus, a depth limit may cause a failure on a solvable problem.

4.2.2 Time limit

By default, the planner runs until it either finds a solution or exhausts the search space. If it takes too long, the user may enter a keyboard interrupt to terminate the execution. Alternatively, she may specify a time limit; then, the system interrupts the search upon reaching this limit. The user may also bound the number of expanded nodes, which causes an interrupt upon reaching the given node number.

4.2.3 Search strategies

The planner normally uses depth-first search and terminates upon finding any solution. The user has two options for changing this default behavior. First, PRODIGY allows breadth-first exploration; however, it is usually much less efficient than the default strategy. Moreover, some heuristics and learning modules do not work with breadth-first search.

Second, the user may ask for *all* solutions to a given problem. Then, the planner explores the entire search space and outputs all available solutions, until it exhausts the space, gets a keyboard interrupt, or reaches a time limit. The system also allows search for an *optimal* solution. This strategy is similar to the search for all solutions; however, when finding a new solution, the system reduces the cost bound and then looks only for better plans. If the planner gets an interrupt, it outputs the best solution found by that time.

4.3 Control rules

The efficiency of depth-first search depends on the heuristics for selecting appropriate branches of the search space, as well as on the order of exploring these branches. The PRODIGY archi-

(a) Select Rule

If (truck-at <to>) is the current subgoal
and **leave-town**(<from>,<to>) is used to achieve it
and (truck-at <place>) holds in the current state
and <place> is of type Town
Then select instantiating <from> with <place>

(b) Reject Rule

If (truck-at <place>) is a subgoal
and (in-truck <pack>) is a subgoal
Then reject the subgoal (truck-at <place>)

Figure 25: Examples of control rules, which encode domain-specific heuristics. The user may provide rules that represent her knowledge about the domain. Moreover, the system includes several mechanisms for the automatic construction of control heuristics.

tecture provides a general mechanism for specifying search heuristics in the form of control rules. These rules usually encode domain-specific knowledge, but they may also represent domain-independent techniques.

A *control rule* is an *if-then* rule that specifies branching decisions, which may depend on the current state, subgoals, and other features of the current plan, as well as on the global state of the search space. The PRODIGY language provides a mechanism for hand-coding control rules. In addition, the architecture includes several learning mechanisms for automatic generation of domain-specific rules.

The system uses three rule types, called select, reject, and prefer rules. A *select rule* points out appropriate branches of the search space. When its applicability conditions match the current plan, the rule generates one or more promising choices. For example, consider the control rule in Figure 25(a). When the planner uses the **leave-town** operator for moving the truck to some destination, the rule indicates that the truck should go there directly from its current location.

A *reject rule* identifies wrong choices and prunes them from the search space. For instance, the rule in Figure 25(b) indicates that, if PRODIGY has to load the truck and drive it to a certain place, then it should delay driving until after loading. Finally, a *prefer rule* specifies the order of exploring branches without pruning any of them. For example, we may replace the select rule in Figure 25(a) with an identical prefer rule, which would mean that the system should first try going directly from the truck's current location to the destination, and it should keep the other options open for later consideration. For some problems, this rule is more appropriate than the more restrictive select rule.

At every decision point, the planner identifies all applicable rules and uses them to make appropriate choices. First, it uses an applicable select rule to choose candidate branches of the search space. If the current plan matches several select rules, the system arbitrarily chooses one of them. If no select rules are applicable, all available branches become candidates. Then, PRODIGY applies all reject rules that match the current plan, and prunes every candidate branch indicated by at least one of these rules. Note that select and reject rules sometimes prune branches that lead to a solution, and they may prevent the system from solving some problems.

After pruning inappropriate branches, PRODIGY applies prefer rules to determine the order of exploring the remaining branches. If the system has no applicable prefer rules, or the rules contradict each other, then it relies on general heuristics for the exploration order.

If the system uses numerous control rules, matching of their conditions at every decision point may take significant time, which sometimes defeats the benefits of the right selection [Minton, 1990]. Wang [1992] designed several techniques that improve the matching efficiency; however, the study of the trade-off between matching time and search reduction remains an open problem.

5 Completeness

A planner is *complete* if it finds a solution for every solvable problem. This notion does not involve a time limit, which means that an algorithm may be complete even if it takes an impractically long time. Although researchers used PRODIGY in multiple studies of learning and search, the question of its completeness had remained unanswered for several years, until Veloso demonstrated the incompleteness of PRODIGY4 in 1995.

We have further investigated completeness issues and found that all bidirectional planners have been incomplete. We have identified specific reasons for their incompleteness and implemented a complete planner by extending PRODIGY search. We have compared it with the incomplete system and demonstrated that the extended planner is almost as efficient as PRODIGY and solves a wider range of problems.

We have already shown that PRODIGY1 and PRODIGY2 do not interleave subgoals and sometimes fail to solve simple problems (see Figure 10). NOLIMIT, PRODIGY4, and FLECS use a more flexible strategy, and their incompleteness arises less frequently. Veloso and Stone [1995] proved the completeness of FLECS using simplifying assumptions, but their assumptions hold only for a limited class of domains.

The incompleteness of bidirectional planners is not a major handicap. Since the search space of most problems is very large, its complete exploration is not feasible, which makes any planner "practically" incomplete. If incompleteness comes up only in a fraction of problems, it is a fair payment for efficiency.

If we achieve completeness without compromising efficiency, we get two benefits. First, we ensure that the planner solves every problem whose search space is sufficiently small for complete exploration. Second, incompleteness may occasionally rule out a simple solution to a large-scale problem, causing an extensive search instead of an easy win. If a planner is complete, it does not rule out any solutions and finds this simple solution.

The incompleteness of means-ends analysis in PRODIGY comes from two sources. First, the planner does not add operators for achieving preconditions that are true in the current state. Intuitively, it ignores potential troubles until they actually arise. Sometimes, it is too late, and the planner fails because it did not take measures earlier. Second, PRODIGY ignores the conditions of if-effects that do not establish any subgoals. Sometimes, such effects negate preconditions of other operators, which may cause a failure.

We achieve completeness by adding new branches to the search space. The main challenge is to minimize the number of new branches in order to preserve efficiency. We describe a method for identifying the crucial branches, based on the information learned in failed old branches, and present an extended planner (Sections 5.1 and 5.2). We believe that this method may prove useful for developing complete versions of other search algorithms.

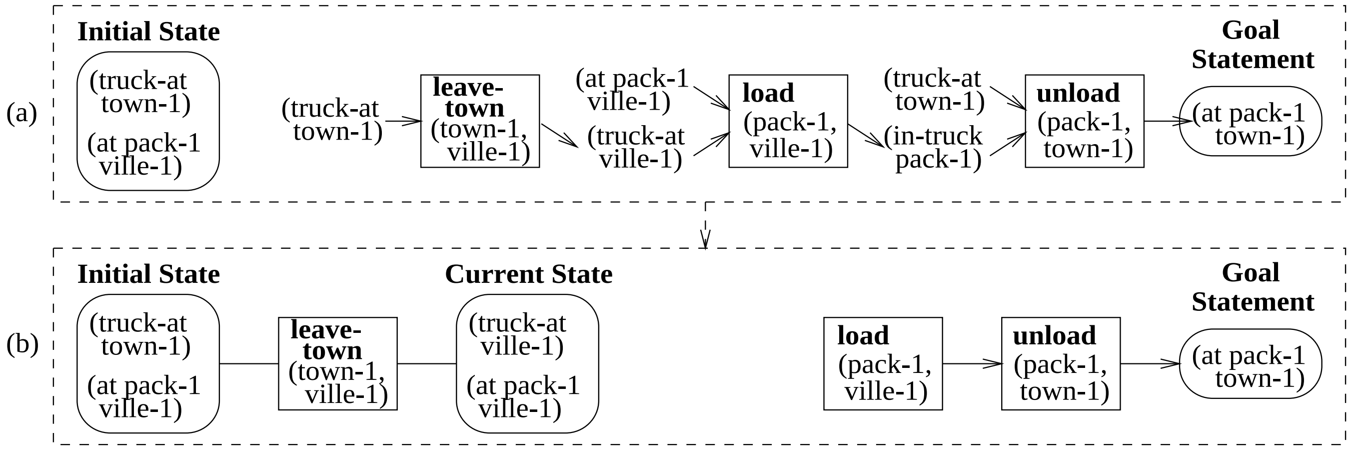


Figure 26: Incompleteness of means-ends analysis in PRODIGY. The planner does not consider fueling the truck before applying **leave-town**(town-1,ville-1). Since the truck cannot leave ville-1 without extra fuel, PRODIGY fails to find a solution.

The full domain language of PRODIGY has two features that aggravate the completeness problem (Section 5.3), which are not addressed in the extended planner. First, eager inference rules may mislead the goal-directed reasoner and cause a failure. Second, functional types allow the reduction of every computational task to a PRODIGY problem, and some tasks are undecidable.

We prove that the extended planner is complete for domains that have no eager inference rules and functional types (Section 5.4). Then, we give experimental results on the performance of the extended planner (Section 5.5). We conclude with a summary of the main results (Section 5.6).

5.1 Limitation of PRODIGY means-ends analysis

GPS, PRODIGY1, and PRODIGY2 were incomplete because they did not explore all branches in their search space. The incompleteness of later means-ends analysis planners has a deeper reason; specifically, they do not try to achieve tail-plan preconditions that hold in the current state.

For example, suppose that the truck is in town-1, pack-1 is in ville-1, and the goal is to get pack-1 to town-1. The only operator that achieves the goal is **unload**(pack-1,town-1), so PRODIGY begins by adding it to the tail-plan (see Figure 26a). The precondition (truck-at town-1) of **unload** is true in the initial state. The planner may achieve the other precondition, (in-truck pack-1), by adding **load**(pack-1,ville-1). The precondition (at pack-1 ville-1) of **load** is true in the initial state, and the other precondition is achieved by **leave-town**(town-1,ville-1), as shown in Figure 26(a).

Now all preconditions are satisfied, and the planner's only choice is to apply **leave-town** (Figure 26b). The application leads into an inescapable trap, where the truck is stranded in ville-1 without fuel. The planner may backtrack and consider different instantiations of **load**, but they will eventually lead to the same trap.

To avoid such traps, a planner must sometimes add operators for achieving literals that

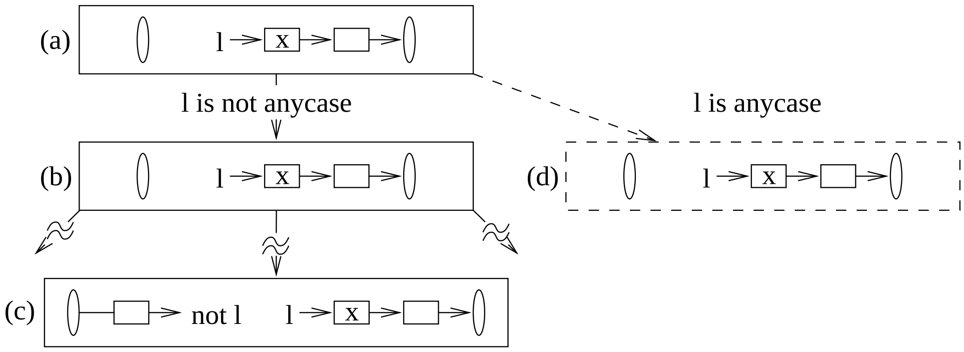


Figure 27: Identifying an anycase subgoal. When PRODIGY adds a new operator x (a), the preconditions of x are not anycase subgoals (b). If some application negates a precondition l of x (c), the planner marks l as anycase and expands the corresponding new branch of the search space (d).

are true in the current state and have not been linked with any tail-plan operators. Such literals are called *anycase subgoals*. The challenge is to identify anycase subgoals among the preconditions of tail-plan operators.

A simple method is to view all preconditions as anycase subgoals. Veloso and Stone [1995] considered this approach in building a complete version of FLECS; however, it caused an explosion in the number of subgoals, leading to gross inefficiency.

Kambhampati and Srivastava [1996b] used a similar approach to ensure the completeness of the Universal Classical Planner. Their system may add operators for achieving preconditions that are true in the current state, if these preconditions are not explicitly linked to the corresponding literals of the state. Although this approach is more efficient than viewing all preconditions as anycase subgoals, it considerably increases branching and often makes the search impractically slow.

A more effective solution is to use information learned in failed branches of the search space. Let us look again at Figure 26. The planner fails because it does not add any operator to achieve the precondition (truck-at town-1) of **unload**, which is true in the initial state. PRODIGY tries to achieve this precondition only when the application of **leave-town** has negated it; however, after the application, the precondition can no longer be achieved.

We see that means-ends analysis may fail when some precondition is true in the current state, but it is later negated by an operator application. We use this observation to identify anycase subgoals: *a precondition or a top-level goal literal is an anycase subgoal if, at some point of search, an application negates it.*

We illustrate a technique for identifying anycase subgoals in Figure 27. Suppose that the planner adds an operator x, with a precondition l, to the tail-plan (plan a in Figure 27). The system creates the branch where l is not an anycase subgoal (plan b). If an application of some operator negates l and if it was true before the application, then l is marked as anycase (plan c). If the planner fails to find a solution in this branch, it eventually backtracks to plan a. If l is marked as anycase, the system creates a new branch, where l is an anycase subgoal (plan d).

If several preconditions of x are marked as anycase, the planner creates the branch where they all are anycase subgoals. During the exploration of this new branch, the algorithm may

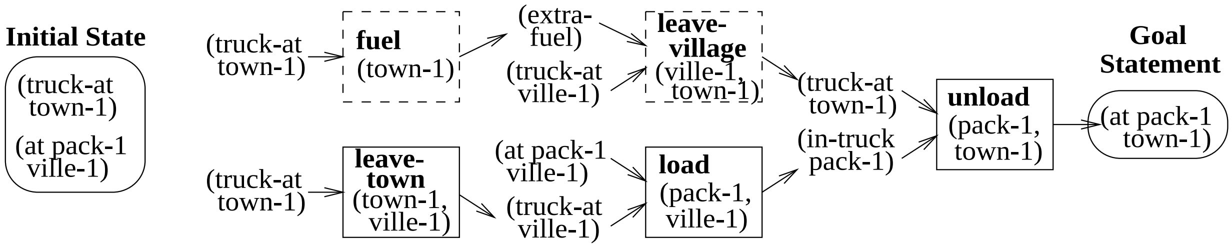


Figure 28: Achieving the subgoal (truck-at town-1), which is satisfied in the current state. First, the planner constructs the three-operator tail-plan shown by solid rectangles. Then, it applies **leave-town** and marks the precondition (truck-at town-1) of **unload** as an anycase subgoal. Finally, it backtracks and adds the two dashed operators to achieve this subgoal.

mark other preconditions of x as anycase. If it again backtracks to plan a, then it creates a branch where the newly marked preconditions are also anycase subgoals.

We now show how this mechanism works for the example problem. The planner first assumes that the preconditions of **unload**(pack-1,town-1) are not anycase subgoals. It builds the tail-plan shown in Figure 26 and applies **leave-town**, negating the precondition (truck-at town-1) of **unload**. The planner then marks this precondition as anycase.

Eventually, the planner backtracks, creates the branch where (truck-at town-1) is an anycase subgoal, and uses **leave-village**(ville-1,town-1) to achieve it (see Figure 28). Then, it constructs the tail-plan shown in Figure 28, which leads to the solution "**fuel**(town-1), **leave-town**(town-1,ville-1), **load**(pack-1,ville-1), **leave-village**(ville-1,town-1), **unload**(pack-1,town-1)."

When the planner identifies the set of all satisfied links (Section 4.1), it does not include anycase links; hence, it never ignores the tail-plan operators that support anycase links. For example, consider the tail-plan in Figure 28; the anycase precondition (truck-at town-1) of **unload** holds in the state, but the planner does not ignore the operators that support it.

We also have to modify the detection of goal loops, described in Section 4.1. For instance, consider again Figure 28; the precondition (truck-at town-1) of **fuel** makes a loop with the identical precondition of **unload**, but the planner should not backtrack. Since this precondition of **unload** is an anycase subgoal, it must not cause goal-loop backtracking. We use Figure 21 to generalize this rule; specifically, if the precondition l of x is an anycase subgoal, then the identical precondition of z does not make a goal loop.

5.2 Clobbers among if-effects

We illustrate the other source of incompleteness, the use of if-effects, in Figure 29. The goal is to load fragile pack-1 without breaking it. The planner adds **load**(pack-1,town-1) to achieve (in-truck pack-1). The preconditions of **load** and the goal "not (broken pack-1)" hold in the current state (Figure 29a), and the planner's only choice is to apply **load**. The application causes the breakage of pack-1 (Figure 29b), and no further planning improves the situation. The planner may try other instantiations of **load**, but they also break the package.

The problem arises because an effect of **load** negates the goal "not (broken pack-1)"; we call it a *clobber effect*. The application reveals the clobber, and the planner backtracks and tries

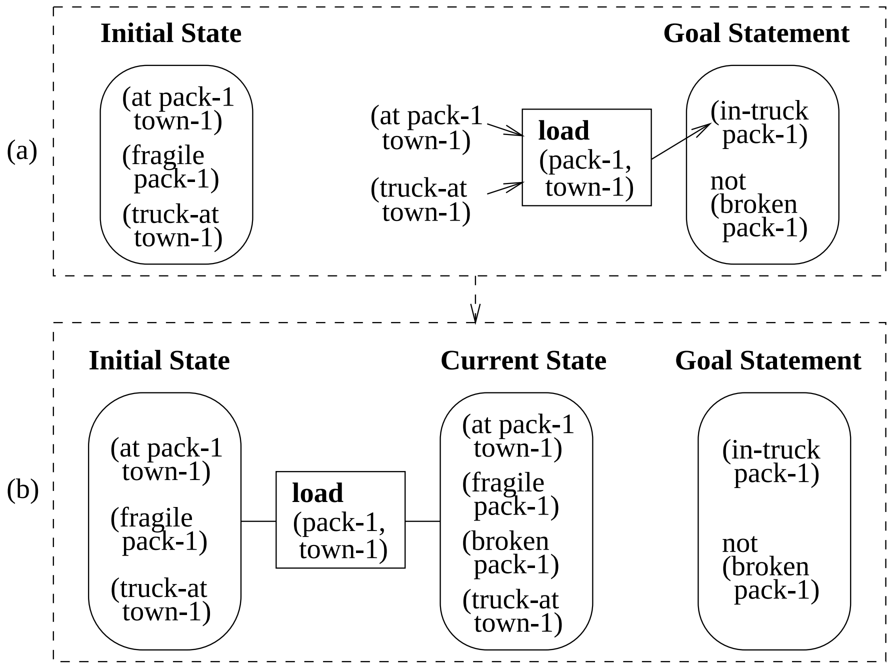


Figure 29: Failure because of a clobber effect. The application of the **load** operator results in breaking the package, and no further actions can undo this damage.

to find an operator that does not cause clobbering. If the clobber effect has no conditions, backtracking is the only way to remedy the situation.

If the clobber is an if-effect, we can try to negate its conditions [Pednault, 1988a; Pednault, 1988b], which may or may not be a good choice; perhaps, it is better to apply the clobber and then re-achieve the negated subgoal. For example, if we had a means for repairing a broken package, we could use it instead of cushioning. We thus need a new decision point, where the planner determines whether it should negate a clobber's conditions.

Introducing this new decision point for every if-effect will ensure completeness, but it may considerably increase branching. We avoid this problem by identifying potential clobbers among if-effects. We detect them in the same way as anycase subgoals; that is, *an if-effect is marked as a potential clobber if it actually deletes some subgoal in one of the failed branches.* The deleted subgoal may be a goal literal, an operator precondition, or a condition of an if-effect that achieves another subgoal. Thus, we again use information learned in failed branches.

We illustrate the mechanism for identifying clobbers in Figure 30. Suppose that the planner adds an operator x with an if-effect e to the tail-plan, and that this operator is added for the sake of its other effects (plan a in Figure 30); that is, e is not linked to a subgoal. Initially, the planner does not try to negate e's conditions (plan b). If x is applied and its effect e negates a subgoal that was true before the application, then the planner marks e as a potential clobber (plan c). If the planner fails to find a solution in this branch, it backtracks to plan a. If e is now marked as a clobber, the planner adds the negation of e's conditions, *cond*, to the operator's preconditions (plan d). If an operator has several if-effects, the planner uses a separate decision point for each of them.

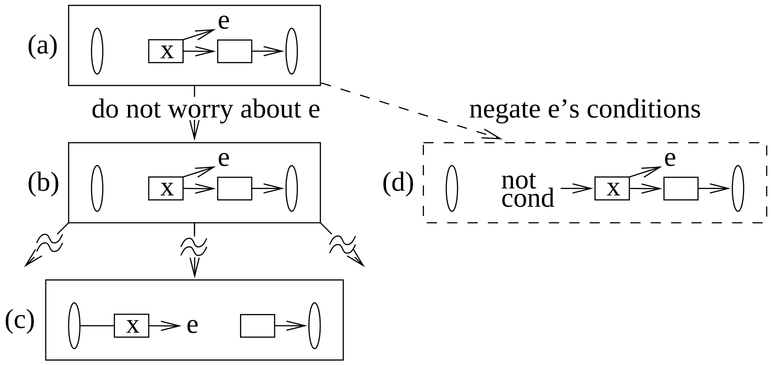


Figure 30: Identifying a clobber effect. When the planner adds a new operator x (a), it does not try to negate the conditions of x's if-effects (b). When applying x, PRODIGY checks whether if-effects negate any subgoals that were true before the application (c). If an if-effect e of x negates some subgoal, the planner marks e as a clobber and adds the corresponding branch to the search space (d).

In the example of Figure 29, the application of **load**(pack-1,town-1) negates the goal "not (broken pack-1)," and the planner marks the if-effect of **load** as a potential clobber (see Figure 31). Upon backtracking, the planner adds the negation of the clobber's condition (fragile pack-1) to the preconditions of **load**. It uses **cushion** to achieve this new precondition and generates the plan "**cushion**(pack-1), **load**(pack-1,town-1)."

We have implemented the extended search algorithm, called RASPUTIN[1], which achieves anycase subgoals and negates the conditions of clobbers. Its main difference from PRODIGY4 is the backward-chaining procedure, summarized in Figure 32. We show its decision points in Figure 33, where thick lines mark the points absent in PRODIGY.

5.3 Other violations of completeness

The PRODIGY system has two other sources of incompleteness, which arise from advanced features of the domain language. We have not addressed them, and their use may violate the completeness of the extended algorithm.

5.3.1 Eager inference rules

If a domain includes eager rules, the planner may fail to solve simple problems. For example, consider the rules in Figure 14 and the problem in Figure 34(a), and suppose that **add-truck-in** is an eager rule. The truck is initially in town-1, within county-1, and the goal is to leave this county.

Since the preconditions of **add-truck-in** hold in the initial state, the system applies it at once and adds (truck-in county-1), as shown in Figure 34(b). If the planner applied

[1]The Russian mystic Grigori Rasputin used the biblical parable of the Prodigal Son to justify his debauchery. He tried to make the story of the Prodigal Son as complete as possible, which is similar to our goal. Furthermore, his name comes from the Russian word *rasputie*, which means *decision point*.

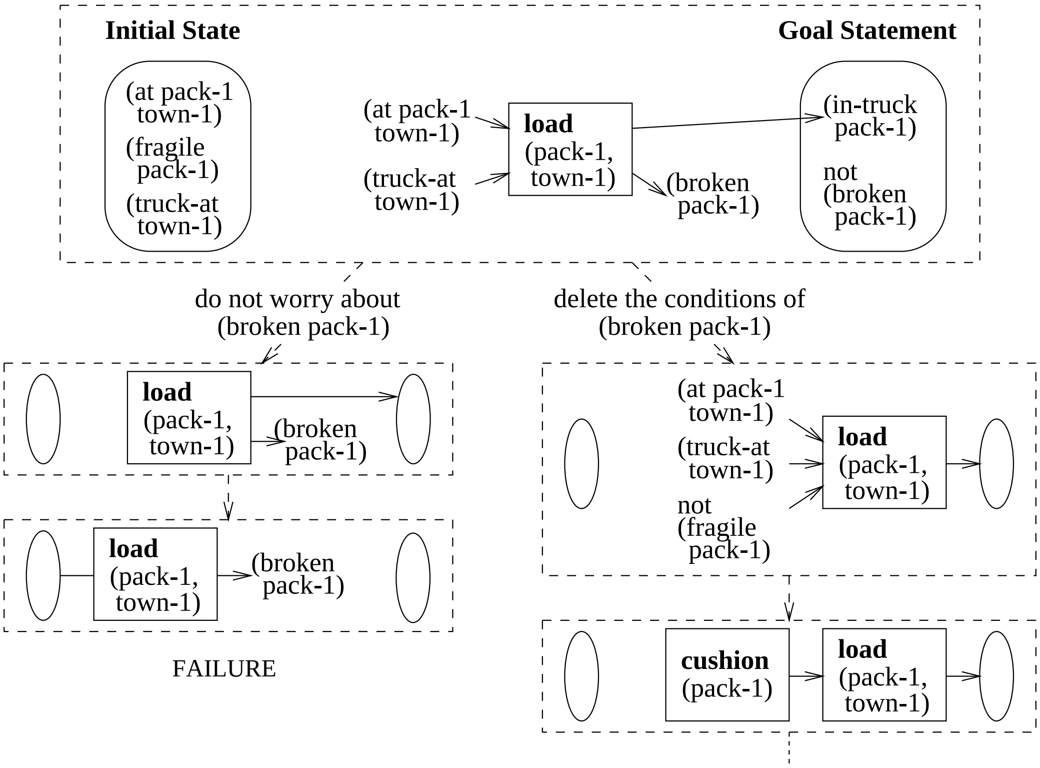


Figure 31: Negating a clobber. When **load** is applied, its if-effect deletes one of the goal literals. The planner backtracks and adds **cushion**, which negates the conditions of this if-effect.

leave-town(town-1,town-2), it would negate the rule's preconditions and delete (truck-in county-1); thus, it would immediately solve the problem.

The planner does not find this solution because it inserts new operators only when they achieve some subgoal, whereas the effects of **leave-town** do not match the goal statement. Since the domain has no operators with a matching effect, the planner terminates with failure. To summarize, the planner sometimes has to negate the preconditions of eager rules that have clobber effects, but it does not consider this option.

5.3.2 Functional types

We next show that functional types enable the user to encode every computational decision task as a PRODIGY problem. Consider the artificial domain and problem in Figure 35. If the object obj satisfies the test function, the planner uses the operator **op**(obj) to achieve the goal; else, the problem has no solution.

The test function may be any Lisp program, and it has access to all data in the PRODIGY architecture. This flexibility allows the user to encode any decision problem, including undecidable and semi-decidable tasks, such as the halting problem. Thus, we may use functional types to specify undecidable PRODIGY problems, and the corresponding completeness issues are beyond the classical planning.

RASPUTIN-**Backward-Chainer**

1c. Pick a literal l among the current subgoals.
Decision point: Choose one of the subgoal literals.

2c. Pick an operator or inference rule *step* that achieves l.
Decision point: Choose one of such operators and rules.

3c. Add *step* to the tail-plan and establish a link from *step* to l.

4c. Instantiate the free variables of *step*.
Decision point: Choose an instantiation.

5c. If the effect that achieves l has conditions,
then add them to *step*'s preconditions.

6c. Use data from the failed branches to identify anycase preconditions of *step*.
Decision point: Choose anycase subgoals among the preconditions.

7c. If *step* has if-effects not linked to l, then:
use data from the failed branches to identify clobber effects;
add the negations of their conditions to the preconditions.
Decision point(s): For every clobber, decide whether to negate its conditions.

Figure 32: Backward-chaining procedure of the RASPUTIN planner. It includes new decision points (lines 6c and 7c), which ensure completeness of PRODIGY means-ends analysis.

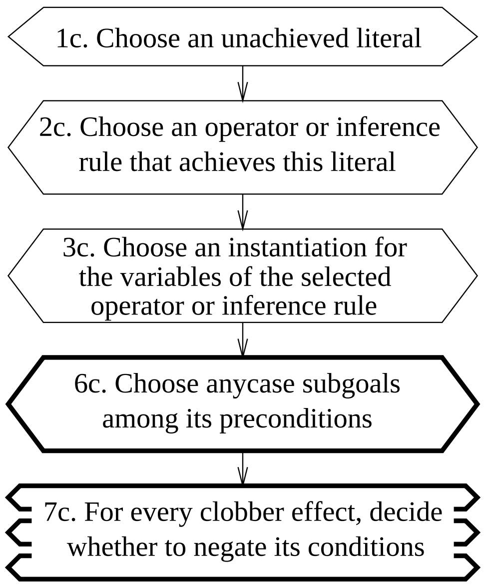
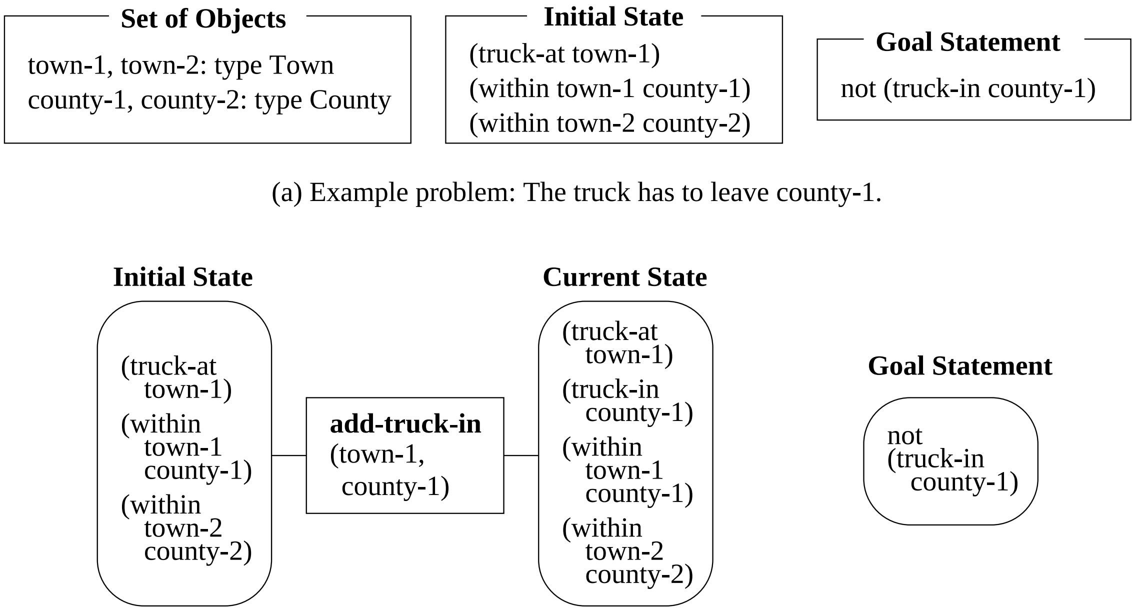


Figure 33: Decision points of RASPUTIN's backward-chaining procedure, given in Figure 32. Thick lines show the decision points that differentiate it from PRODIGY (see Figures 8 and 9).



(a) Example problem: The truck has to leave county-1.

(b) Application of an inference rule.

Figure 34: Failure because of an eager inference rule. PRODIGY does not negate the preconditions of the rule **add-truck-in**(town-1,county-1), which clobbers the goal.

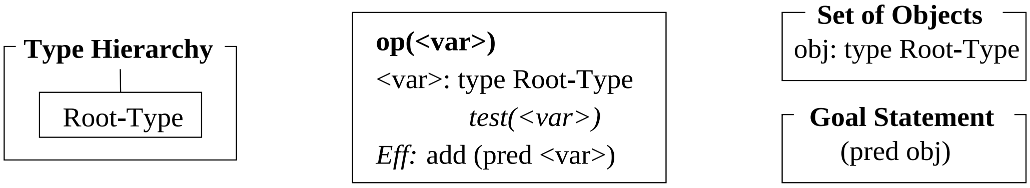


Figure 35: Reducing an arbitrary decision task to a PRODIGY problem. The test function is a Lisp procedure that encodes the decision task. Since functional types allow encoding of undecidable problems, they cause incompleteness of PRODIGY search.

5.4 Completeness proof

If a domain has no eager inference rules or functional types, the extended planner is complete. To prove it, we show that, for every solvable problem, some sequence of choices in the planner's decision points leads to a solution.

Suppose that a problem has a solution "$step_1, step_2, step_3, ..., step_n$," where every step is either an operator or a lazy inference rule, and that no other solution has fewer steps. We begin by defining clobber effects, subgoals, and justified effects in the solution plan.

A *clobber* is an if-effect such that (1) its conditions do not hold in the plan and (2) if we applied its actions anyways, they would make the plan incorrect.

A *subgoal* is a goal literal or precondition such that either (1) it does not hold in the initial state or (2) it is negated by some prior operator or inference rule. For example, a precondition of $step_3$ is a subgoal if either it does not hold in the initial state or it is negated by $step_1$. Every subgoal in a solution is achieved by some operator or inference rule; for

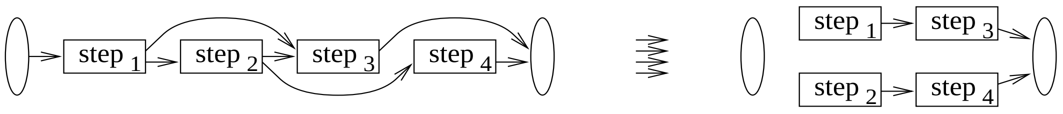


Figure 36: Converting a shortest solution into a tail-plan. We link every subgoal to the last operator or inference rule that achieves it, and use these links to construct a tree-structured tail-plan.

example, if $step_1$ negates a precondition of $step_3$, then $step_2$ must achieve it.

A *justified effect* is the last effect that achieves a subgoal or negates a clobber's conditions. For example, if $step_1$, $step_2$, and $step_3$ all achieve some subgoal precondition of $step_4$, then the corresponding effect of $step_3$ is justified, since it is the last among the three.

If a condition literal in a justified if-effect does not hold in the initial state, or if it is negated by some prior step, we consider it a subgoal. Note that the definition of such a subgoal is recursive: we define it through a justified effect, and a justified effect is defined in terms of a subgoal in some step that comes after it.

Since we consider a shortest solution, each step has at least one justified effect. If we link each subgoal and each clobber's negation to the corresponding justified effect, we may use the resulting links to convert the solution into a tree-structured tail-plan, as illustrated in Figure 36. If a step is linked to several subgoals, we use any one of these links in the tail-plan.

We now show that the extended algorithm can construct this tail-plan. If no subgoal holds in the initial state and the solution has no clobber effects, the tail-plan construction is straightforward. The nondeterministic algorithm creates the desired tail-plan by always calling *Backward-Chainer* rather than applying operators, choosing subgoals that correspond to the links of the desired solution (see Line 1c in Figure 32), selecting the appropriate operators and inference rules (Line 2c), and generating the right instantiations (Line 4c).

If some subgoal literal holds in the initial state, the planner first builds a tail-plan that has no operator linked to this subgoal. Then, the application of some step negates the literal, and the planner marks it as an anycase subgoal. It can then backtrack to the point *before the first application* and choose the right operator or inference rule for achieving the subgoal. Similarly, if the plan has a clobber effect, the algorithm can detect it by applying operators and inference rules. It can then backtrack to the point before the applications and add the right step for negating the clobber's conditions. Note that, even if the planner always makes the right choice, it may have to backtrack for every subgoal that holds in the initial state and for every clobber effect.

Eventually, the algorithm constructs the plan consisting of the desired tail-plan and no head-plan. It can then produce the solution by always deciding to apply, rather than adding new tail-plan operators, and selecting applicable steps in the right order.

5.5 Performance of the extended planner

We have tested RASPUTIN in three domains and compared it with PRODIGY4. We present data on the relative efficiency of the two planners and show that RASPUTIN solves more problems than PRODIGY4.

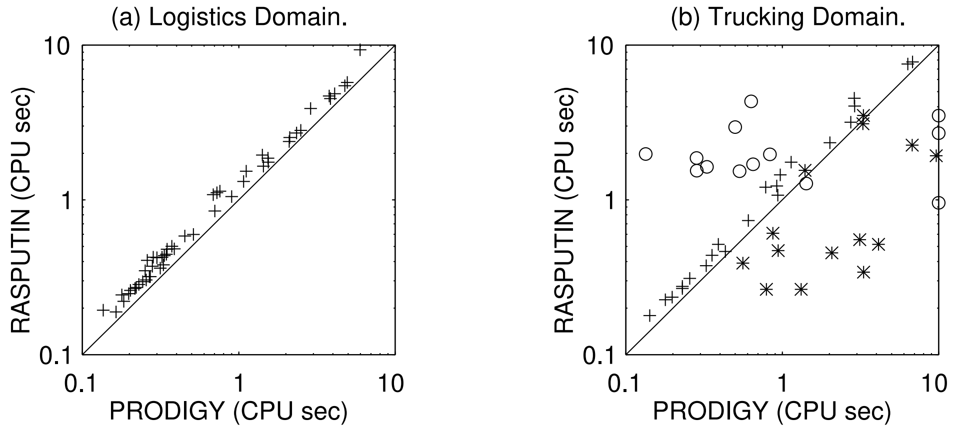


Figure 37: Comparison of RASPUTIN and PRODIGY in the (a) Logistics Domain and (b) Trucking Domain. The horizontal axes give the search times of PRODIGY, whereas the vertical axes show the efficiency of RASPUTIN on the same problems. Pluses (+) and asterisks (∗) mark the problems solved by both planners, whereas circles (o) are the ones solved only by RASPUTIN.

We first give results for the PRODIGY Logistics Domain [Veloso, 1994]; the task is to construct plans for transporting packages by vans and airplanes. The domain consists of several cities, each of which has an airport and postal offices. We use airplanes for carrying packages between airports, and vans for delivery within cities. This domain has no if-effects and does not give rise to situations that require achieving anycase subgoals; thus, PRODIGY4 performs better than the complete planner.

We ran both planners on fifty problems, which differed in the number of cities, vans, airplanes, and packages. We randomly generated initial locations of packages, vans, and airplanes, as well as destinations of packages. The results are summarized in Figure 37(a), where each plus (+) denotes a problem. The horizontal axis shows PRODIGY's running times, and the vertical axis gives RASPUTIN's times for the same problems. Since PRODIGY is faster on all problems, all pluses are above the diagonal. The ratio of RASPUTIN's to PRODIGY's time varies from 1.20 to 1.97; its mean is 1.45.

We have run similar tests in the PRODIGY Process-Planning Domain [Gil, 1991], which also does not require negating if-effect conditions or achieving anycase subgoals. The task is to construct plans for making mechanical parts with specified properties, using available machining equipment. The ratio of RASPUTIN's to PRODIGY's time in this domain is between 1.22 and 1.89, with the mean at 1.39.

We next show results in an extended version of the Trucking Domain. We now use multiple trucks and connect towns and villages by roads. A truck can go from one place to another only if there is a road between them. We have experimented with different numbers of towns, villages, trucks, and packages. We have randomly generated road connections, initial locations of trucks and packages, and destinations of packages.

In Figure 37(b), we show the performance of PRODIGY and RASPUTIN on fifty problems. The twenty-two problems denoted by pluses (+) do not require clobber negation or anycase subgoals. PRODIGY outperforms RASPUTIN on these problems, with a mean ratio of 1.27.

The fourteen problems denoted by asterisks (∗) require the use of anycase subgoals or the negation of clobbers' conditions for finding an efficient solution, but can be solved inefficiently without it. RASPUTIN wins on twelve of these problems and loses on two. The ratio of PRODIGY's to RASPUTIN's time varies from 0.90 to 9.71, with the mean at 3.69. This ratio grows with the number of required anycase subgoals.

Finally, the circles (o) show the sixteen problems that cannot be solved without anycase subgoals and the negation of clobbers. PRODIGY hits the 10-second time limit on some problems and terminates with failure on the others, whereas RASPUTIN solves all of them.

5.6 Summary of completeness results

We have developed a new bidirectional planner, by extending PRODIGY search; the resulting algorithm is complete for a subset of the PRODIGY domain language. The full language includes two features that may violate completeness: eager inference rules and functional types. To our knowledge, the extended algorithm is the first complete planner that uses means-ends analysis. It is about 1.5 times slower than PRODIGY4 on the problems that do not require negating clobbers' conditions and planning for anycase subgoals; however, it solves problems that PRODIGY cannot solve.

We have built the extended planner in three steps. First, we have identified the specific reasons for incompleteness of previous systems. Second, we have added new decision points to eliminate these reasons, without significant increase of the search space. Third, we have implemented an algorithm that explores the branches of the old search space first, and extends the space only after failing to find a solution in the old space. We conjecture that this three-step approach may prove useful for enhancing other incomplete planners.

Acknowledgments

We are grateful to Manuela Veloso and Henry Rowley for their valuable comments and suggestions. The work was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and Defense Advanced Research Project Agency (DARPA) under grant number F33615-93-1-1330.

References

[Blum and Furst, 1997] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

[Blythe and Reilly, 1993a] Jim Blythe and W. Scott Reilly. Integrating reactive and deliberative planning in a household robot. In *Proceedings of the* AAAI *Fall Symposium on Instantiating Real-World Agents*, pages 6–13, 1993.

[Blythe and Reilly, 1993b] Jim Blythe and W. Scott Reilly. Integrating reactive and deliberative planning for agents. Technical Report CMU-CS-93-155, School of Computer Science, Carnegie Mellon University, 1993.

[Blythe and Veloso, 1992] Jim Blythe and Manuela M. Veloso. An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 13–19, 1992.

[Borrajo and Veloso, 1996] Daniel Borrajo and Manuela M. Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review*, 10:1–34, 1996.

[Carbonell and Gil, 1990] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation: The operator refinement method. In Ryszard S. Michalski and Yves Kodratoff, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 3, pages 191–213. Morgan Kaufmann, San Mateo, CA, 1990.

[Carbonell *et al.*, 1991] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*, pages 241–278. Lawrence Erlbaum Associates, Mahwah, NJ, 1991.

[Carbonell *et al.*, 1992] Jaime G. Carbonell, Jim Blythe, Oren Etzioni, Yolanda Gil, Robert L. Joseph, Dan Kahn, Craig A. Knoblock, Steven Minton, Alicia Pérez, W. 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.

[Carbonell, 1983] Jaime G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371–392. Tioga Publishers, Palo Alto, CA, 1983.

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

[Cox and Veloso, 1997a] Michael T. Cox and Manuela M. Veloso. Controlling for unexpected goals when planning in a mixed-initiative setting. In Ernesto Costa and Amilcar Cardoso, editors, *Progress in Artificial Intelligence: Eighth Portuguese Conference on Artificial Intelligence*, pages 309–318. Springer-Verlag, Berlin, Germany, 1997.

[Cox and Veloso, 1997b] Michael T. Cox and Manuela M. Veloso. Supporting combined human and machine planning: An interface for planning by analogical reasoning. In David B. Leake and Enric Plaza, editors, *Case-Based Reasoning Research and Development: Second International Conference on Case-Based Reasoning*, pages 531–540. Springer-Verlag, Berlin, Germany, 1997.

[Etzioni, 1990] Oren Etzioni. *A Structural Theory of Explanation-Based Learning*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Technical Report CMU-CS-90-185.

[Etzioni, 1993] Oren Etzioni. A structural theory of explanation-based learning. *Artificial Intelligence*, 60(1):93–140, 1993.

[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(3–4):189–208, 1971.

[Fink and Veloso, 1996] Eugene Fink and Manuela M. Veloso. Formalizing the PRODIGY planning algorithm. In Malik Ghallab and Alfredo Milani, editors, *New Directions in* AI *Planning*, pages 261–271. IOS Press, Amsterdam, Netherlands, 1996.

[Fink, 2003] Eugene Fink. *Changes of Problem Representation: Theory and Experiments*. Springer Verlag, Berlin, Germany, 2003.

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

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

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. Technical Report CMU-CS-92-175.

[Golding *et al.*, 1987] Andrew R. Golding, Paul S. Rosenbloom, and John E. Laird. Learning general search control from outside guidance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 334–337, 1987.

[Haigh and Veloso, 1997] Karen Zita Haigh and Manuela M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. In *Proceedings of the First International Conference on Autonomous Agents*, pages 363–370, 1997.

[Haigh and Veloso, 1998a] Karen Zita Haigh and Manuela M. Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1):79–95, 1998.

[Haigh and Veloso, 1998b] Karen Zita Haigh and Manuela M. Veloso. Planning, execution and learning in a robotic agent. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 120–127, 1998.

[Haigh, 1998] Karen Zita Haigh. *Situation-Dependent Learning for Interleaved Planning and Robot Execution*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998. Technical Report CMU-CS-98-108.

[Joseph, 1992] Robert L. Joseph. *Graphical Knowledge Acquisition for Visually-Oriented Domains*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. Technical Report CMU-CS-92-188.

[Kambhampati and Srivastava, 1996a] Subbarao Kambhampati and Biplav Srivastava. Unifying classical planning approaches. Technical Report 96-006, Department of Computer Science, Arizona State University, 1996.

[Kambhampati and Srivastava, 1996b] Subbarao Kambhampati and Biplav Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In Malik Ghallab and Alfredo Milani, editors, *New Directions in* AI *Planning*, pages 61–75. IOS Press, Amsterdam, Netherlands, 1996.

[Knoblock, 1993] Craig A. Knoblock. *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning.* Kluwer Academic Publishers, Boston, MA, 1993.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

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

[Minton *et al.*, 1989a] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Dan R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40(1–3):63–118, 1989.

[Minton *et al.*, 1989b] 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 *et al.*, 1994] Steven Minton, John Bresina, and Mark Drummond. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research*, 2:227–262, 1994.

[Minton, 1988] Steven Minton. *Learning Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers, Boston, MA, 1988.

[Minton, 1990] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2–3):363–391, 1990.

[Newell and Simon, 1961] Allen Newell and Herbert A. Simon. GPS, a program that simulates human thought. In Heinz Billing, editor, *Lernende Automaten*, pages 109–124. R. Oldenbourg Verlag, Munich, Germany, 1961.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice Hall, Upper Saddle River, NJ, 1972.

[Pednault, 1988a] Edwin P. D. Pednault. Extending conventional planning techniques to handle actions with context-dependent effects. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 55–59, 1988.

[Pednault, 1988b] Edwin P. D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial-order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 103–114, 1992.

[Pérez and Carbonell, 1993] M. Alicia Pérez and Jaime G. Carbonell. Automated acquisition of control knowledge to improve the quality of plans. Technical Report CMU-CS-93-142, School of Computer Science, Carnegie Mellon University, 1993.

[Pérez and Etzioni, 1992] M. Alicia Pérez and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 367–372, 1992.

[Pérez, 1995] M. Alicia Pérez. *Learning Search Control Knowledge to Improve Plan Quality.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995. Technical Report CMU-CS-95-175.

[Pohl, 1971] Ira Pohl. Bi-directional search. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 6*, pages 127–140. American Elsevier Publishers, New York, NY, 1971.

[Stone and Veloso, 1996] Peter Stone and Manuela M. Veloso. User-guided interleaving of planning and execution. In Malik Ghallab and Alfredo Milani, editors, *New Directions in* AI *Planning*, pages 103–112. IOS Press, Amsterdam, Netherlands, 1996.

[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 Artificial Intelligence Planning Systems*, pages 164–169, 1994.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[Veloso and Blythe, 1994] Manuela M. Veloso and Jim Blythe. Linkability: Examining casual link commitments in partial-order planning. In *Proceedings of the Second International Conference on Artificial Intelligence 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, 1994.

[Veloso and Carbonell, 1990] Manuela M. Veloso and Jaime G. Carbonell. Integrating analogy into a general problem-solving architecture. In Maria Zemankova and Zbigniew W. Ras, editors, *Intelligent Systems*, pages 29–51. Ellis Horwood, Chichester, United Kingdom, 1990.

[Veloso and Carbonell, 1993a] Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278, 1993.

[Veloso and Carbonell, 1993b] Manuela M. Veloso and Jaime G. Carbonell. Towards scaling up machine learning: A case study with derivational analogy in PRODIGY. In Maria Zemankova and Zbigniew W. Ras, editors, *Machine Learning Methods for Planning*, pages 233–272. Morgan Kaufmann, San Mateo, CA, 1993.

[Veloso and Stone, 1995] Manuela M. Veloso and Peter Stone. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, 3:25–52, 1995.

[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 *et al.*, 1997] Manuela M. Veloso, Alice M. Mulvehill, and Michael T. Cox. Rationale-supported mixed-initiative case-based planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 1072–1077, 1997.

[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, Berlin, Germany, 1994.

[Wang, 1992] Xuemei Wang. Constraint-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 Artificial Intelligence Planning Systems*, pages 335–340, 1994.

[Wang, 1996] Xuemei Wang. *Learning Planning Operators by Observation and Practice.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Technical Report CMU-CS-96-154.

[Warren, 1974] David H. D. Warren. WARPLAN: A system for generating plans. Technical Report Memo 76, Department of Computational Logic, University of Edinburgh, 1974.

[Weld, 1994] Daniel S. Weld. An introduction to least commitment planning. *Artificial Intelligence Magazine*, 15(4):27–61, 1994.

[Yang *et al.*, 1996] Qiang Yang, Josh Tenenberg, and Steve Woods. On the implementation and evaluation of ABTWEAK. *Computational Intelligence*, 12:307–330, 1996.