Prodigy search

Newell and Simon [1961; 1972] developed the *means-ends analysis* technique 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 systems [Fikes and Nilsson, 1971; Warren, 1974; Tate, 1977] gradually abandoned forward search and began to rely exclusively on backward chaining.

Researchers investigated several types of backward chainers [Minton *et al.*, 1994] and discovered 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 problem solvers.

Meanwhile, PRODIGY researchers extended means-ends analysis and designed a family of problem solvers based on the combination of goal-directed backward chaining with simulation of operator execution. The underlying strategy is a special case of bidirectional search [Pohl, 1971]. It has given rise to several versions of the PRODIGY system, including PRODIGY1, PRODIGY2, NOLIMIT, PRODIGY4, and FLECS.

The developed algorithms keep track of the domain state that results from executing parts of the currently constructed solution, and use the state to guide the goal-directed reasoning. Least commitment proved ineffective for this search technique, and Veloso developed an alternative strategy, based on instantiating all variables as early as possible.

Experiments have demonstrated that PRODIGY search is an efficient procedure, a fair match to least-commitment systems and other successful problem solvers. Moreover, the PRODIGY architecture has proved a valuable tool for the development of learning techniques, and researchers have used it in constructing a number of systems for the automated acquisition of control knowledge.

We have utilized this architecture in the work on the representation changes and constructed the $\mathcal{S}$HAPER system as an extension to PRODIGY. In particular, $\mathcal{S}$HAPER's library of problem solvers is based on PRODIGY search algorithms. We therefore describe these algorithms before presenting $\mathcal{S}$HAPER.

First, we review the past work on the PRODIGY system and discuss advantages and drawbacks of the developed search techniques (Section 2.1). Then, we describe the foundations of these techniques and their use in different versions of PRODIGY (Section 2.2), as well as

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, Perez, and Gil
FLECS	1994	Veloso and Stone

Table 2.1: Main versions of the PRODIGY architecture. The work on this problem-solving architecture continued for over ten years, and gave rise to a series of novel search strategies.

the main extensions to the basic search engine (Sections 2.3 and 2.4). Finally, we report the results of a joint investigation with Blythe on the completeness of the PRODIGY search technique (Section 2.5).

2.1 PRODIGY system

The PRODIGY system went through several stages of development, over the course of ten years, and gradually evolved into an advanced architecture, which supports a variety of search and learning techniques. We give a brief history of its development (Section 2.1.1) and summarize the main features of the resulting search engines (Section 2.1.2).

2.1.1 History

The history of the PRODIGY architecture (see Table 2.1) began circa 1986, when Minton and Carbonell implemented PRODIGY1, which became a testbed for their work on control rules [Minton, 1988; Minton *et al.*, 1989a]. They concentrated on explanation-based learning of control knowledge and left few records of the original search engine.

Minton, Carbonell, Knoblock, and Kuokka used PRODIGY1 as a prototype in their work on PRODIGY2 [Carbonell *et al.*, 1990], which supported an advanced language for describing problem 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.

Veloso [1989] and Borrajo developed the next version, called NOLIMIT, which significantly differed from its predecessors. In particular, they added new branching points, which made the search near-complete, 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 large-scale transportation problems.

Blythe, Wang, Veloso, Kahn, Pérez, and Gil developed a collection of techniques for enhancing the search engine and built PRODIGY4 [Carbonell *et al.*, 1992]. In particular, they provided an efficient technique for instantiating operators [Wang, 1992], extended the use of inference rules, and designed advanced data structures to improve the efficiency of the low-level implementation. They also implemented a friendly user interface and tools for adding new learning mechanisms.

Veloso and Stone [1995] implemented the FLECS algorithm, an extension to PRODIGY4 that included an additional decision point and new strategies for exploring the search space, and demonstrated that their strategies often improved the efficiency.

The PRODIGY architecture provides ample opportunities for the application of speed-up learning, and researchers have used it to develop and test a variety of techniques for the automated efficiency improvement. Minton [1998] designed the first learning module for the PRODIGY architecture, which automatically generated control rules. He demonstrated the effectiveness of integrating learning with PRODIGY search, which stimulated work on other efficiency-improving techniques.

In particular, researchers have 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 problem-solving 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 solutions [Pérez and Carbonell, 1993; Pérez, 1995], learning unknown properties of the problem 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]. These research results have been major contributions to the study of machine learning; however, they have left two notable gaps. First, PRODIGY researchers tested the learning modules *separately,* without exploring their synergetic use. Even though preliminary attempts to integrate learning with abstraction gave positive results [Knoblock *et al.*, 1991a], the researchers have not pursued this direction. Second, there has been no automated techniques for deciding when to invoke specific learning modules. The user has traditionally been responsible for the choice among available learning systems. Addressing these gaps is among the goals of our work on the $\mathcal{S}$HAPER system.

2.1.2 Advantages and drawbacks

The PRODIGY architecture is based on two major design decisions, which differentiate it from other problem-solving systems. First, it combines backward chaining with simulated execution of relevant operators. Second, it fully instantiates operators in early stages of search, whereas most classical systems delay the commitment to a specific instantiation.

The backward-chaining procedure selects operators relevant to the goal, instantiates them, and arranges them into a partial-order solution. The forward chainer simulates the execution of these operator and gradually constructs a total-order sequence of operators. The system keeps track of the simulated world state, which would result from executing this sequence.

The problem solver utilizes the simulated world state in selecting operators and their instantiations, which improves the effectiveness of goal-directed reasoning. In addition, PRODIGY learning modules use the state to identify reasons for successes and failures of the search algorithm.

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

On the flip side, early commitment to full instantiations and specific execution order leads to a large branching factor, which results in gross inefficiency of breadth-first search. The problem solver uses depth-first search and relies on heuristics for selecting appropriate branches of the search space, which usually leads to finding suboptimal solutions. If the heuristics prove misleading, the solver expands wrong branches and may fail to find a solution. When a problem has no solution, a large branching factor becomes a major handicap: PRODIGY cannot exhaust the available space in reasonable time.

A formal comparison of PRODIGY with other search systems is still an open problem; however, multiple experimental studies have 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 standard 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 combines them with direct forward search. They implemented the Universal Classical Planner (UCP), which can use all these search strategies; however, the resulting general algorithm has many branching points, which give rise to an impractically large search space. The main open problem is development of heuristics that would effectively use the flexibility of UCP to guide the search.

Blum and Furst [1997] constructed GRAPHPLAN, which uses the domain state in a different way. They implemented propagation of constraints from the initial state of the domain, which enables their system to identify some operators with unsatisfiable preconditions. The system then discards these operators and uses backward chaining to construct a solution from the remaining operators. GRAPHPLAN performs forward constraint propagation prior to the search for a solution. Unlike PRODIGY, it does *not* use forward search from the initial state.

The relative performance of PRODIGY and GRAPHPLAN also varies from domain to domain. The GRAPHPLAN algorithm has to generate and store all possible instantiations of all operators before searching for a solution, which often causes a combinatorial explosion; thus, PRODIGY usually faster than GRAPHPLAN in large-scale domains. On the other hand, GRAPHPLAN wins in small-scale domains that require extensive search.

Researchers recently 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 routes of a household robot in a simulated environment. Stone and Veloso [1996] constructed a mechanism for user-guided interleaving of problem solving and execution.

Haigh and Veloso [1996; 1997; 1998a; 1998b] built a system that navigates XAVIER, a real robot at Carnegie Mellon University. Haigh [1998] integrated this system with XAVIER's low-level control procedures, and demonstrated its effectiveness in planning and guiding the robot's high-level actions.

Their interleaving algorithms begin the real-world execution before PRODIGY completes the search for a solution, thus eliminating some backtracking points in the search space. This strategy involves the risk of bringing the robot to a deadend or even into an inescapable trap. To avoid such traps, Haigh and Veloso restricted the use of their system to domains with reversible actions.

2.2 Search engine

We next describe the basics of PRODIGY search; the description is based on the results of joint work with Veloso on formalizing the main principles underlying the PRODIGY system [Fink and Veloso, 1996]. All versions of the system are based on the algorithm described here; however, they differ from each other in the decision points used for backtracking, and in the general heuristics for guiding the search.

We present the foundations of the PRODIGY domain language (Section 2.2.1), encoding of intermediate incomplete solutions (Section 2.2.2), and the algorithm that combines backward chaining with execution simulation (Sections 2.2.3 and 2.2.4). We delay the discussion of techniques for handling disjunctive and quantified preconditions until Section 2.3. After describing the search engine, we discuss differences among the main versions of PRODIGY (Section 2.2.5).

2.2.1 Encoding of problems

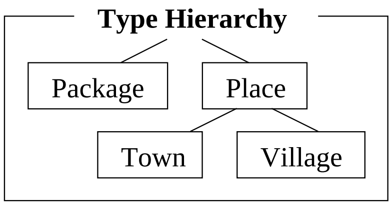
We define a *problem domain* by a set of object types and a library of operators that act on objects of these types. The PRODIGY language for describing operators is based on the STRIPS domain language [Fikes and Nilsson, 1971], extended to express conditional effects, disjunctive preconditions, and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

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

We may specify conditional effects, also called *if-effects,* whose outcome depends on the domain state. An if-effect is defined by its *conditions* and *actions*. If the conditions hold, the effect changes the state, according to its actions. Otherwise, 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, then the operator cannot be executed. On the other hand, if the conditions of an if-effect do not hold, we may execute the operator, but the if-effect does not change the state.

The actions of an if-effect are predicates, to be added to or deleted from the state; that is, their encoding is identical to that of unconditional effects. We refer to both unconditional effects and if-effect actions as *simple effects.* When we talk about "effects" without explicitly referring to if-effects, we mean simple effects.




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

```
leave-village(<from>, <to>)
<from>: type Village
<to>: type Place
Pre: (truck-at <from>)
     (extra-fuel)
Eff: del (truck-at <from>)
     add (truck-at <to>)
```

```
load(<pack>, <place>)
<pack>: type Package
<place>: type Place
Pre: (at <pack> <place>)
     (truck-at <place>)
Eff: del (at <pack> <place>)
     add (in-truck <pack>)
     (if (fragile <pack>)
       add (broken <pack>))
```

```
unload(<pack>, <place>)
<pack>: type Package
<place>: type Place
Pre: (in-truck <pack>)
     (truck-at <place>)
Eff: del (in-truck <pack>)
     add (at <pack> <place>)
```

```
fuel(<place>)
<place>: type Town
Pre: (truck-at <place>)
Eff: add (extra-fuel)
```

```
cushion(<pack>)
<pack>: type Package
Eff: del (fragile <pack>)
```

Figure 2.1: 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.

In Figure 2.1, we give an example of a simple domain. Note that the syntax of this domain description slightly differs from the PRODIGY language [Carbonell *et al.*, 1992], for the purpose of better readability. The domain includes two types of objects, Package and Place. 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.

We have to load packages before driving to their destination and unload afterwards. If a package is fragile, it gets broken during loading. We may cushion a package by soft material, which removes the fragility and prevents breakage.

A *problem* is defined by a list of object instances, an *initial state*, and a *goal statement*. The initial state is a set of literals, whereas the goal statement is a condition that must hold after executing a solution. A *complete solution* is a sequence of instantiated operators that can be executed from the initial state to achieve the goal. We give an example of a problem in Figure 2.2. The task in this problem 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))."

The initial state may include literals that cannot be added or deleted by operators, called *static literals*. For example, if the domain did not include the **fuel** operator, then (extra-fuel)

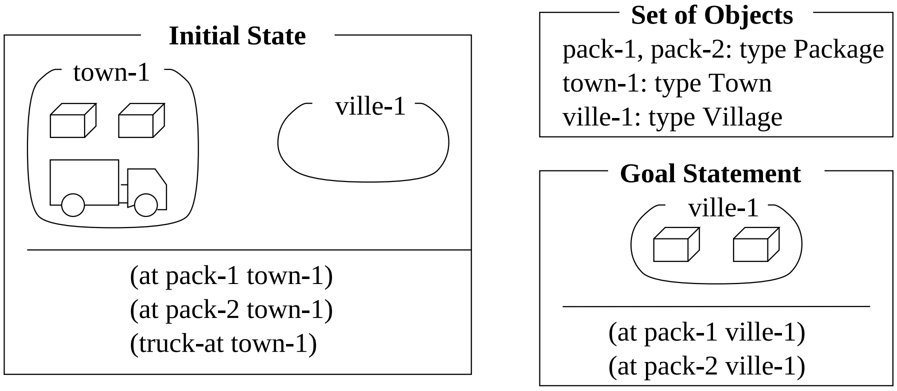


Figure 2.2: Encoding of a problem in the Trucking Domain, which includes a set of object instances, initial world state, and goal statement; the task is to deliver two packages from `town-1` to `ville-1`.

would be a static literal. If all instantiations of a predicate are static literals, we say that the predicate itself is static. Since no operator sequence can affect these literals, the goal statement should be consistent with the static elements of the initial state. Otherwise, the problem is unsolvable and the system reports failure without search.

2.2.2 Incomplete solutions

Given a problem, most problem-solving systems begin with the empty set of operators 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 solutions.* We view them as nodes in the search space of the solver algorithm. Each modification of a current incomplete solution gives rise to a new node, and the number of possible modifications determines the branching factor of the search.

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

In PRODIGY, an incomplete solution consists of two parts, a total-order *head* and tree-structured *tail* (see Figure 2.3). The root of the tail's tree is the goal statement G, the other nodes are fully instantiated operators, and the edges are ordering constraints.

The tail 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, it *instantiates* the operator, that is, replaces all the variables with specific objects. The preconditions of a fully instantiated operator are a conjunction of literals, where every literal is an instantiated predicate.

The head is a sequence of instantiated operators that can be executed from the initial state. It is generated by the execution-simulating algorithm described in Section 2.2.3. The

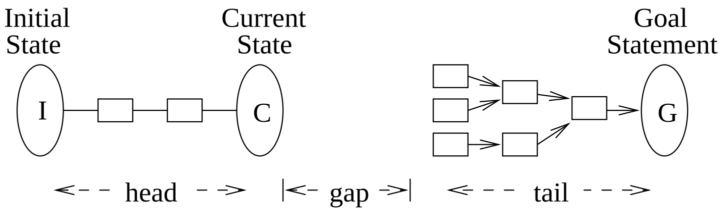


Figure 2.3: Representation of an incomplete solution: It consists of a total-order head, which can be executed from the initial state, and a tree-structured tail constructed by a backward chainer. The current state C is the result of applying the head operators to the initial state I.

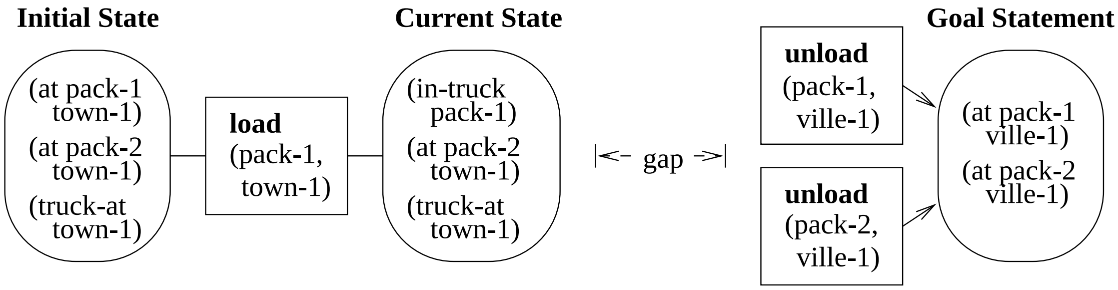


Figure 2.4: Example of an incomplete solution in the Trucking Domain: The head consists of a single operator, **load**; the tail comprises two **unload** operators, linked to the goal literals.

state C achieved by executing the head is the *current state*. In Figure 2.4, we illustrate an incomplete solution for the example trucking problem.

Since the head is a total-order sequence of operators that do not contain variables, the current state C is uniquely defined. The backward chainer, responsible for the tail, views C as its initial state. If the tail operators cannot be executed from the current state C, then there is a "gap" between the head and tail. The purpose of problem solving is to bridge this gap. For example, we can bridge the gap in Figure 2.3 by a sequence of two operators, "**load**(pack-2,town-1), **leave-town**(town-1,ville-1)."

2.2.3 Simulating execution

Given an initial state I and a goal statement G, PRODIGY begins with the empty head and tail, and modifies them, step by step, until it builds a complete solution. Thus, the initial incomplete solution has no operators and its current state is the same as the initial state, $C = I$.

At each step, PRODIGY can modify the current incomplete solution in one of two ways (see Figure 2.5). First, it can add an operator to the tail (operator t in the picture), to achieve a goal literal or a precondition of another operator. Tail modification is a job of the backward-chaining algorithm, described in Section 2.2.4.

Second, PRODIGY can move some operator op from the tail to the head (operator x in the picture). The preconditions of op must be satisfied in the current state C. The operator op becomes the last operator of the head, and the current state is modified according to the effects of op. The search algorithm usually has to select among several operators that can

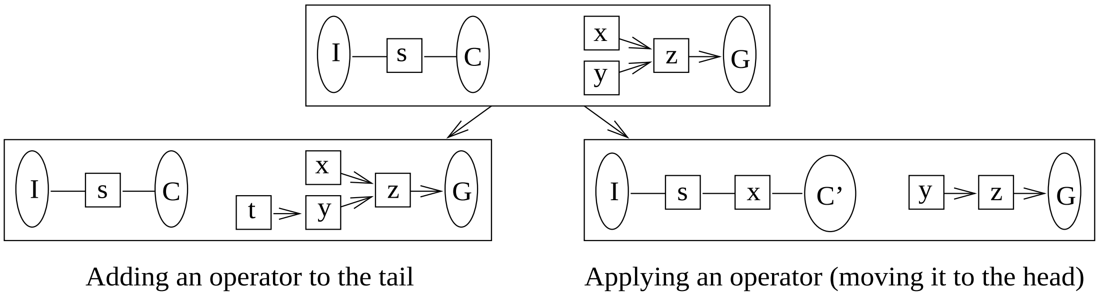


Figure 2.5: Modifying an incomplete solution: PRODIGY either adds a new operator to the tail tree (left), or moves one of the previously added operator to the head (right).

be moved to the head; thus, it needs to decide on the order of executing these operators.

Intuitively, we may imagine that the system executes the head operators in the real world and has already changed the world from its initial state I to the current state C. If the tail contains an operator whose preconditions are satisfied in C, then PRODIGY applies this operator and further changes the state. Because of this analogy with the real-world changes, moving an operator from the tail to the head is called the *application* of the operator; however, this term refers to *simulating* an operator application. Even if the execution of the head operators is disastrous, the world does not suffer: the search algorithm backtracks and tries 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 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 simulates the application of **leave-town**(town-1,town-1), whose effects are "del (truck-at town-1)" and "add (truck-at town-1)." The system first removes this literal from the state description, and then adds it back. If the system processed the effects in the opposite order, it would permanently remove the truck's location, thus obtaining an inconsistent state.

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

2.2.4 Backward chaining

We next describe the backward-chaining procedure that constructs the tree-structured tail of an incomplete solution. When the problem solver invokes this procedure, it adds a new operator to the tail, for achieving either a goal literal or a precondition literal of another tail operator. Then, it establishes a link from the newly added operator to the literal achieved by this operator, and adds the corresponding ordering constraint. For example, if the incomplete solution is as shown in Figure 2.4, then the procedure may add the operator **load**(pack-2,ville-

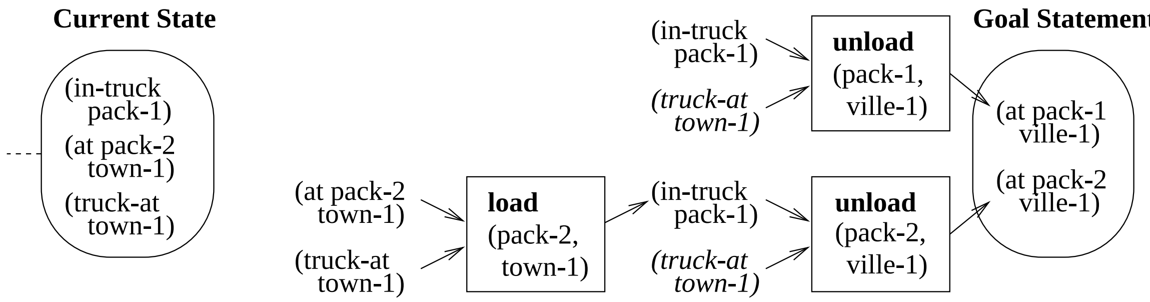


Figure 2.6: Example of the tail in an incomplete solution to a trucking problem. First, the backward chainer adds the **unload** operators, which achieve the two goal literals. Then, it inserts **load** to achieve the precondition (in-truck pack-1) of **unload**(pack-2,ville-1).

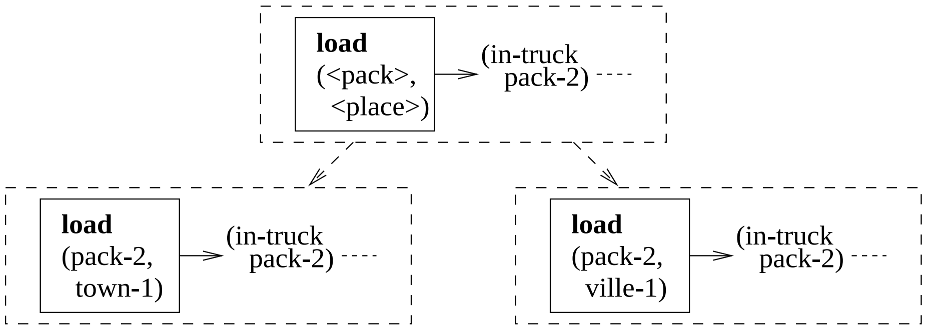


Figure 2.7: Instantiating a newly added operator: If the set of objects is as shown in Figure 2.2, PRODIGY can generate two alternative versions of **load** for achieving the subgoal (in-truck pack-2).

1) to achieve the precondition (in-truck pack-2) of **unload**(pack-2,ville-1) (see Figure 2.6). If the backward chainer uses an if-effect of an operator 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 it is not true in the current state C and has not been linked to any tail operator.* Unsatisfied goal literals and preconditions are called *subgoals.* For example, the tail in Figure 2.6 has two identical subgoals, marked by italics.

Before inserting an operator into the tail, the solver fully instantiates it, that is, substitutes all free variables of the operator with specific object instances. Since the PRODIGY domain language allows the use of 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 2.7). First, PRODIGY instantiates the variable <pack> with the instance pack-2 from the subgoal literal. Then, it has to instantiate the other free 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 2.7).

In Figure 2.8, we summarize the search algorithm, which explores the space of incomplete solutions. The *Operator-Application* procedure builds the head and maintains the current state, whereas *Backward-Chainer* constructs the tail.

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 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 2.9.

Note that the first two choices (Lines 2a and 1b) enable the problem solver to consider different orderings of head operators. These two choices are essential for solving problems with interacting subgoals; they are analogous to the choice of ordering constraints in least-commitment algorithms.

2.2.5 Main versions

The algorithm in Figure 2.9 has five decision points, which allow flexible selection of operators, their instantiations, and order of their execution; however, these decisions give rise to a large branching factor. The use of built-in heuristics, which eliminate some of the available choices, may reduce the search space and improve the efficiency. On the negative side, such heuristics prune some solutions and may direct the search to a suboptimal solution, or even prevent finding any solution. Determining appropriate restrictions on the solver's choices is one of the major research problems.

Even though the described algorithm underlies all PRODIGY versions, from PRODIGY1 to FLECS, the versions differ in their use of decision points and built-in 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. We outline the use of the backtracking mechanism and its evaluation in the PRODIGY architecture.

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].

PRODIGY1 and PRODIGY2

The early versions of PRODIGY had only two backtracking points: the choice of an operator (Line 2c in Figure 2.8) 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 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 2.2. The solver adds **unload**(pack-1,ville-1) to achieve (at pack-1 ville-1), and **load**(pack-1,town-1) to achieve the precondition (in-truck pack-1) of **unload** (see Figure 2.10a). Then, it applies **load** and adds **leave-town**(town-1,ville-1) to achieve the precondition (truck-at ville-1) of unload (Figure 2.10b). Finally, PRODIGY applies **leave-town** and **unload** (Figure 2.10c), thus bringing only one package to the village.

Base-PRODIGY

1a. If the goal statement G is satisfied in the current state C, then return the head.

2a. Either

(i) *Backward-Chainer* adds an operator to the tail,

(ii) or *Operator-Application* moves an operator from the tail to the head.

Decision point: Choose between (i) and (ii).

3a. Recursively call *Base*-PRODIGY on the resulting incomplete solution.

Operator-Application

1b. Pick an operator op, in the tail, such that

(i) there is no operator in the tail ordered before op,

(ii) and 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 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 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 2.8: Foundations of the PRODIGY search algorithm: The *Operator-Application* procedure simulates execution of operators, whereas *Backward-Chainer* selects operators relevant to the goal.

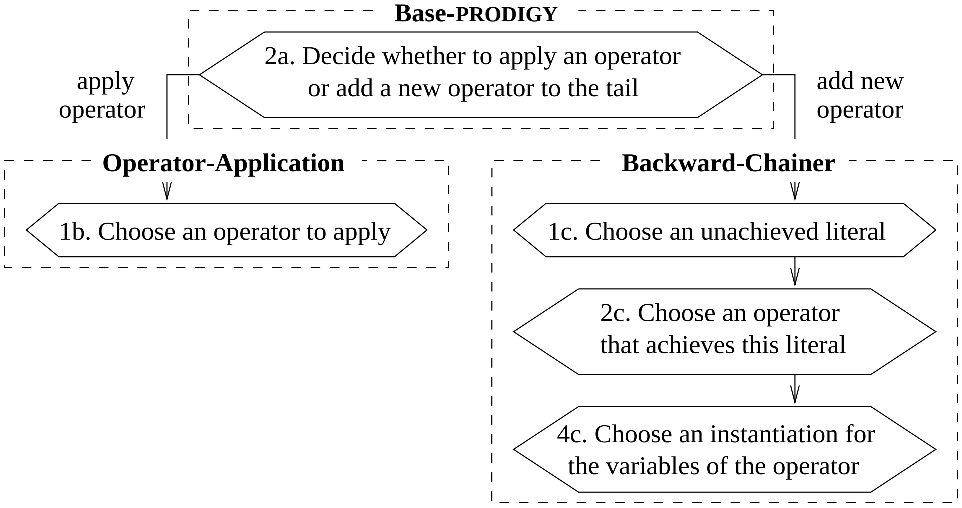


Figure 2.9: Main decision points in the PRODIGY search engine, summarized in Figure 2.8. Every decision point allows backtracking, thus giving rise to multiple branches of the search space.

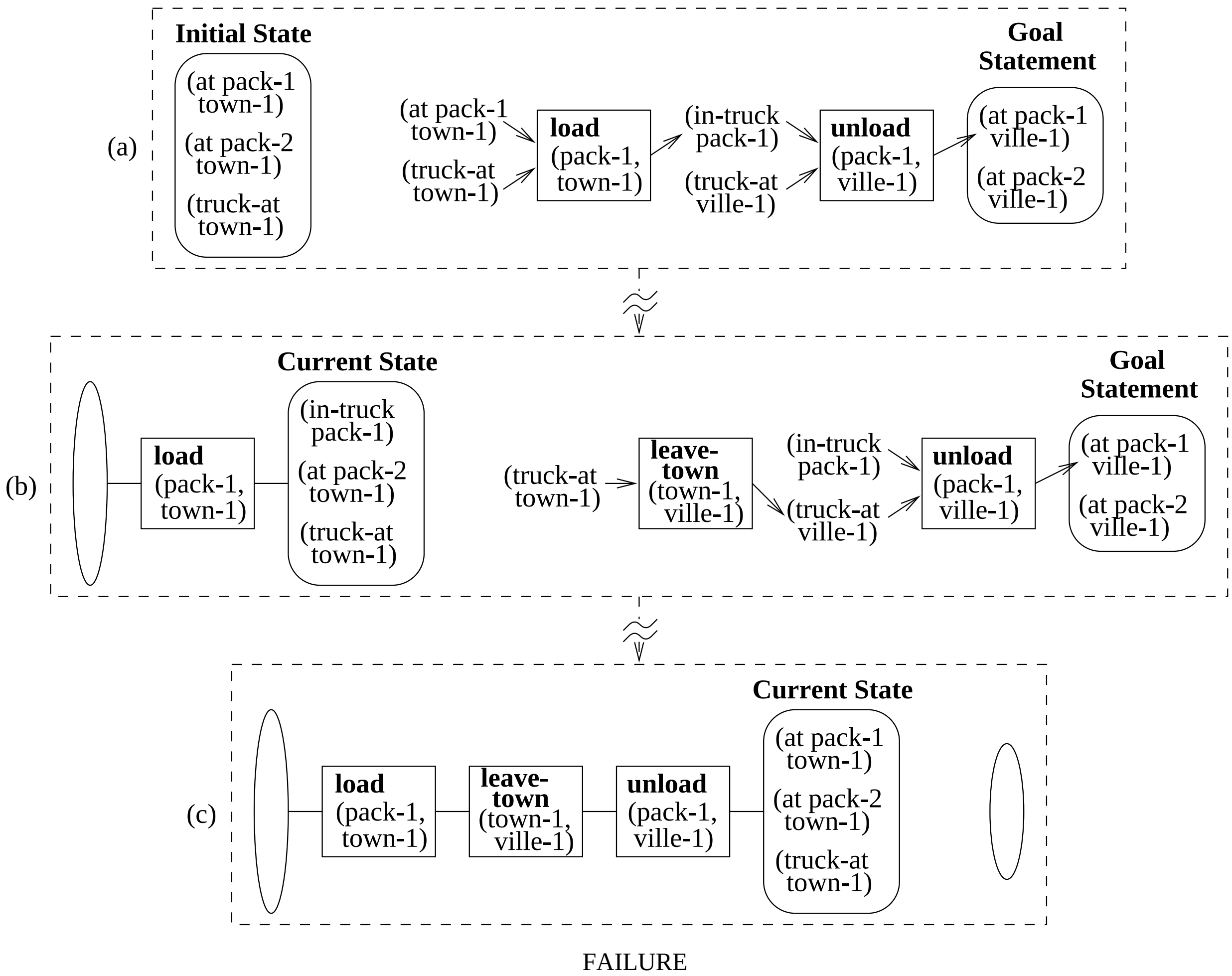


Figure 2.10: Incompleteness of PRODIGY2: The system fails to solve the trucking problem in Figure 2.2. Since the PRODIGY2 search algorithm always prefers the operator application to adding new operators, it cannot load both packages before driving the truck to its goal destination.

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, which 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 allows the enhancement of performance; however, it requires the human operator to assume the responsibility for completeness and solution quality.

NOLIMIT and PRODIGY4

During the work on the NOLIMIT system, Veloso added two more backtracking points, delaying the application of tail operators (Line 2a) and choosing a subgoal (Line 1c), and

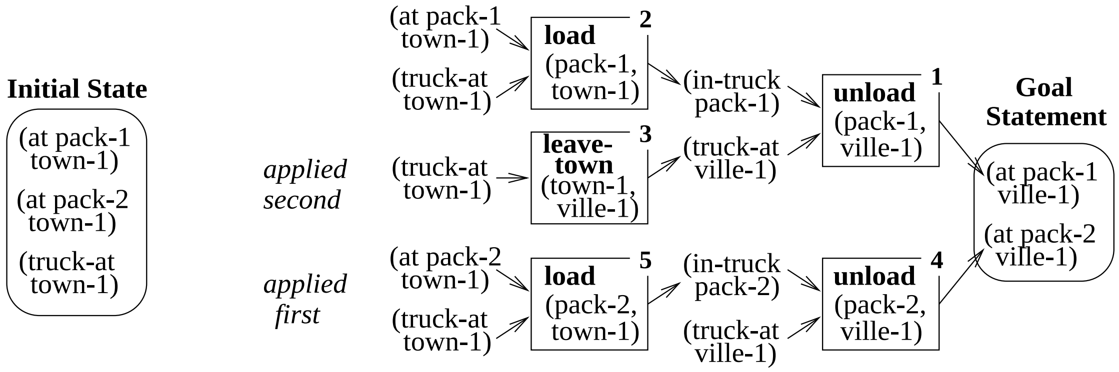


Figure 2.11: Example of inefficiency in the PRODIGY4 system; the boldface numbers, in the upper right corners of operators, mark the order of adding operators to the tail. Since the PRODIGY4 algorithm always applies the last added operator, it attempts to apply **leave-town** before one of the **load** operators, which leads to a deadend and requires backtracking.

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 sometimes negatively affects the search time.

For instance, if we use PRODIGY4 to solve the problem in Figure 2.2, it may generate the tail shown in Figure 2.11, 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 solver cannot use this application order. The system applies **leave-town** before one of the **load** operators, which leads to a deadend. It then has to backtrack and construct a new tail, which allows the right order of applying the operators.

FLECS

The FLECS algorithm 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 algorithm, called SAVTA and SABA, which differ in their choice between adding an operator to the tail and applying an operator (Line 2a). SAVTA prefers to apply tail operators before adding new ones, whereas SABA tries to delay their application.

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

Veloso has recently fixed a minor bug in the implementation of SABA, which sometimes led to inappropriate search decisions; however, she has not yet reported empirical evaluation of the corrected algorithm. Note that we employed the original version of SABA in experiments with the $\mathcal{S}$HAPER system, since the bug has been found after the completion of our work.

cushion(<pack>, <place>)
<pack>: type Package
<place>: type Place
Pre: (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 2.12: Examples of disjunction and quantification in the PRODIGY domain language. The user may utilize these constructs in precondition expressions and goal statements.

2.3 Extended domain language

The PRODIGY domain language is an extension of the STRIPS language [Fikes and Nilsson, 1971]. The STRIPS system used a limited description of operators, and PRODIGY researchers added several advanced features, which allowed encoding of large-scale domains. The new features included complex preconditions and goal expressions (Section 2.3.1), inference rules, (Section 2.3.2), and flexible use of object types (Section 2.3.3).

2.3.1 Extended operators

The PRODIGY domain 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 solution quality.

Disjunctive preconditions

To illustrate the use of disjunction, we consider a variation of the **cushion** operator, given in Figure 2.12(a). In this example, we can cushion a package when it is inside or near the truck.

When PRODIGY instantiates an operator with disjunctive preconditions, it generates an instantiation for one element of the disjunction and discards all other elements. For example, if the solver 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 solver backtracks and considers the instantiation of another element. For instance, if the selected version of the **cushion**(o)perator has proved inadequate, PRODIGYmay discard the first element of the conjunction, `(at <pack> <place>)`, and choose the instantiation `(in-truck pack-1)` of the other element.

Quantified preconditions

We illustrate the use of quantifiers in Figures 2.12(b) and 2.12(c). In the first example, the solver has to transport *any* package to `ville-1`. In the second example, it has to deliver *all* packages to the village.

When the problem solver instantiates an existential quantification, 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 2.12(b) by `(at pack-1 ville-1)`. If the chosen object does not lead to a solution, PRODIGY backtracks and tries another one. When instantiating a universally quantified expression, the solver treats it as a conjunction over all matching objects.

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, the problem solver generates all possible instantiations of its preconditions and chooses one of them. If the solver 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] has designed an advanced algorithm for generating possible instantiations of operators and goal statements. In particular, she developed an efficient mechanism for pruning inconsistent choices of objects and provided heuristics for selecting the most promising instantiations.

Costs

The use of operator costs allows us to measure the quality of complete solutions. We assign nonnegative numerical costs to instantiated operators, and define a solution cost as the sum of its operator costs. The lower the cost, the better the solution.

The authors of the original PRODIGY architecture did not provide support for operator costs, and usually measured the solution quality by the number of operators. Pérez [1995] has implemented a mechanism for using operator costs during her exploration of control rules for improving solution quality; however, she did not incorporate costs into the main version.

We re-implemented the cost mechanism during the work on the SHAPER system. In Figure 2.13, we give an example of cost encoding. For every operator, the user specifies a Lisp function, whose arguments are operator variables. Given specific object instances, the function returns the corresponding cost, which must be a nonnegative real number. If the operator cost does not depend on the instantiation, it may be specified by a number rather than a function. If the user does not encode a cost, then by default it is 1.

The example in Figure 2.13(a) includes two cost functions, called *leave-cost* and *load-cost*. We give pseudocode for these functions (Figure 2.13b) and their real encoding in the PRODIGY system (Figure 2.13c).

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

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; esle, return 3.

(b) Pseudocode of the cost functions.

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

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

(c) Actual LISP functions.

Figure 2.13: Encoding of operator costs: The user may specify a constant cost value or, alternatively, a Lisp function that inputs operator variables and returns a nonnegative real number. If the description of an operator does not include a cost, PRODIGY assumes that it is 1.

literals, and 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 problem solver 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. Note that, since incomplete solutions consist of fully instantiated operators, the solver can determine the cost of every intermediate solution.

2.3.2 Inference rules

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

Example

In Figure 2.14, we show three inference rules for the Trucking Domain. In this example, we have made two modifications to the original domain description (see Figure 2.1). First, the **cushion** operator adds (cushioned <pack>) instead of deleting (fragile <pack>), and the **add-fragile** rule indicates that uncushioned packages are fragile. Thus, the user does not have to specify fragility in the initial state.

Second, the domain includes the type County and the predicate (within <place> <county>). Note that this predicate is static, that is, it is not an effect of any operator or inference rule. 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

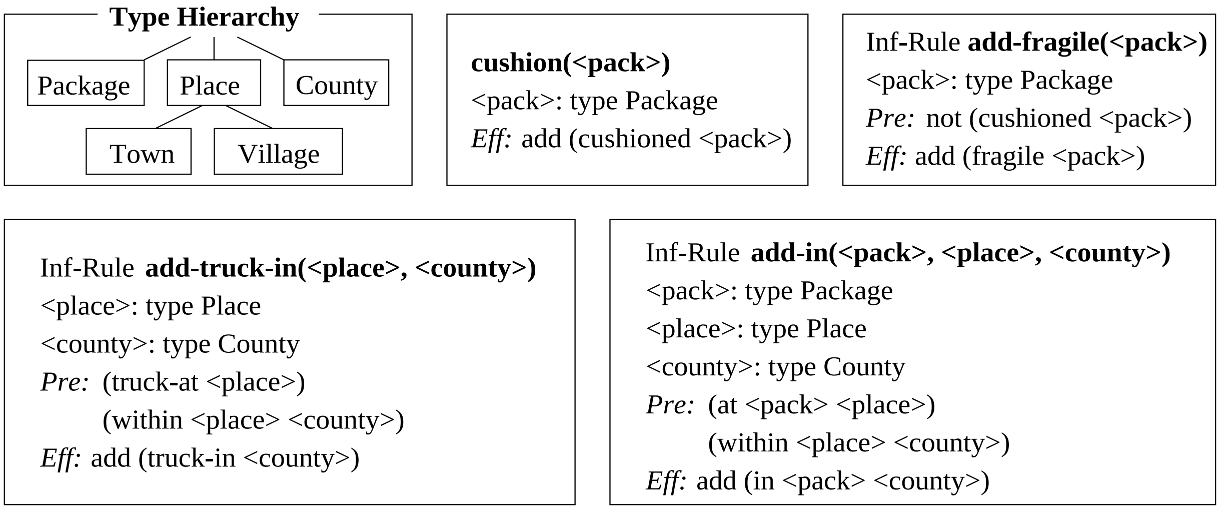


Figure 2.14: Encoding of inference rules in the PRODIGY domain language. These rules point out indirect results of changing the world state; their syntax is identical to that of operators.

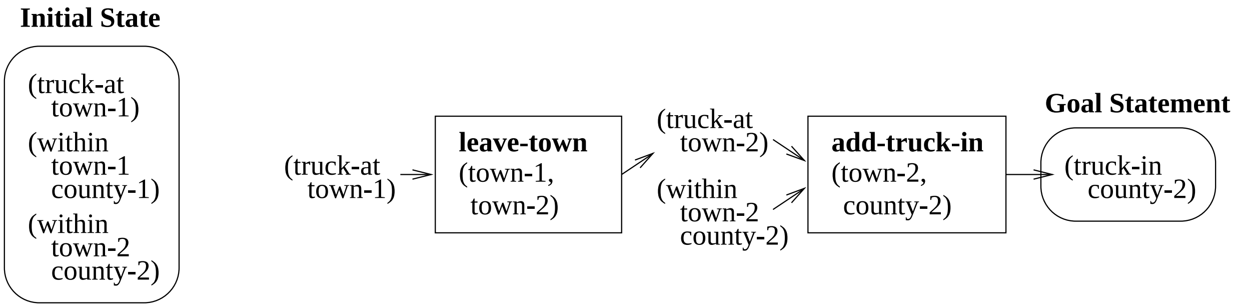


Figure 2.15: Use of an inference rule in backward chaining: PRODIGY links the **add-truck-in** rule to the goal literal, and then adds **leave-town** to achieve the rule's precondition (truck-at town-2).

county-1) to the current state. Similarly, we use add-in to infer the current county of each package.

Use of inferences

The encoding of inference rules is the same as that of operators, which may include disjunctive and quantified preconditions, and if-effects; however, the rules have no costs and their use does not affect the overall solution cost.

The use of inference rules is also similar to that of operators: the problem solver adds an instantiated rule to the tail, for achieving the selected subgoal, and applies the rule when its preconditions hold in the current state. We illustrate it in Figure 2.15, where the solver uses the **add-truck-in** rule to achieve the goal, and then adds **leave-town** to achieve a 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 2.16(a) adds (truck-in town-2) to the state. If the system then applies

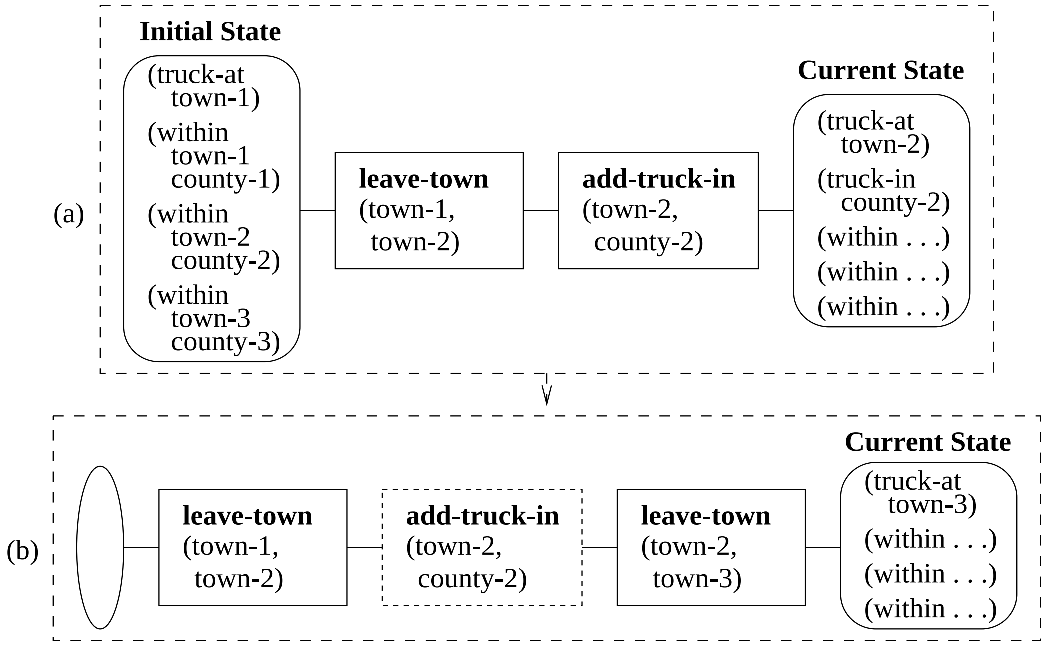


Figure 2.16: Cancelling the effects of an inference rule upon the negation of its preconditions: When PRODIGY applies **leave-town**(town-2,town-3), it negates the precondition (truck-at town-2) of the **add-truck-in** rule; hence, the system removes the effects of this rule from the current state.

leave-town (Figure 2.16b), it negates the preconditions of **add-truck-in** and, hence, cancels its effects. This semantics differs from the use of operators, whose effects remain in the state, unless deleted by opposite effects of other operators.

Eager and lazy rules

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

When the user encodes a domain, she has to mark all rules that have unwanted effects. 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. The marked inference rules are called *eager rules*, whereas the others are *lazy rules.* Note that *Backward-Chainer* may use both eager an lazy rules, and the only special property of eager rules is their forced application in the matching states. If the user wants *Backward-Chainer* to disregard some eager rules, she may provide a control rule that prevents their use in the tail.

Truth maintenance

When the PRODIGY system 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. If some rules that remain in force have if-effects, the system must check the conditions of every if-effect, which may also lead to modification of the state. Next, PRODIGY looks for an eager rule whose conditions hold in the resulting state. If the system finds such a rule, then it applies the rule and further changes the state.

If inference rules interact with each other, then 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 effects of old rules. This chain of state modifications, which does not involve search, is similar to the firing of productions in the Soar system [Laird *et al.*, 1986; Golding *et al.*, 1987].

Blythe designed an efficient truth-maintenance procedure, which keeps track of all applicable inference rules and controls the described forward chaining. The solver invokes this procedure after each application of an operator or inference rule from the tail.

If the user provides inference rules, she has to ensure that the resulting 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 application. The PRODIGY system does *not* check for such inconsistencies, and an inappropriate rule set may cause unpredictable results.

2.3.3 Complex types

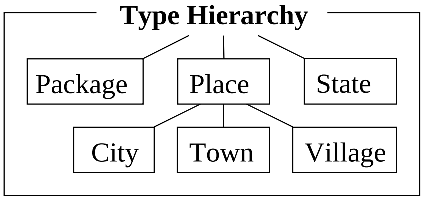
We have already explained the use of a type hierarchy (see Figure 2.1), which defines object classes and enables the user to specify the allowed values of variables in operator preconditions, conditions of if-effects, and goal statements. For example, the possible values of the <from> variable in **leave-town** include all towns, but not villages. The early versions of the PRODIGY system did not support a type hierarchy. Veloso designed a typed domain language during her work on NOLIMIT, and the authors of PRODIGY4 further developed the mechanism for using types.

A type hierarchy is a tree, whose nodes are called *simple types*. For instance, the hierarchy in Figure 2.1 has five simple types: Package, Town, Village, Place, and the root type that includes all objects. We have illustrated the use of simple types in the operator encoding; however, they often do not provide sufficient flexibility.

For example, consider the type hierarchy in Figure 2.17 and suppose that truck may get in 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 defining complex constraints, through disjunctive and functional types.

Disjunctive types

We illustrate the use of a disjunctive type in Figure 2.17, where it specifies the possible values of <from> and <place>. The user specifies a disjunctive type as a set of simple types; in our example, it includes Town and City. When the problem solver instantiates the corresponding variable, it uses an object that belongs to any of these types. For instance, the system may use the **leave-town** operator for departing from a town or city.



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

fuel(<place>)
<place>: type (or Town City)
Pre: (truck-at <place>)
Eff: add (extra-fuel)

Figure 2.17: 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.

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 2.18: 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, which defines the fully connected graph of roads.

Functional types

We give an example of a functional type in Figure 2.18, where it 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 our example, the valid values of the <to> variable include all places that have road connections with the <from> location.

The boolean function is an arbitrary Lisp procedure, whose arguments are the 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.

For instance, if every place is connected with every other place except itself, then we use the test function given in Figure 2.18(b). The domain encoding must include a Lisp implementation of this function, as shown in Figure 2.18(c).

Use of test functions

When the system 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 us to encode functional effects, that is, operator effects that depend on the current state.

Generator functions

The system also supports the use of *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 2.18, we give an example that involves both a test function, called *positive,* and a generator function, *decrement.* In this example, the system keeps track of the available space in the trunk. If there is no space, it cannot load more packages. We use the 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 used with a generator function, she may define a numerical type that includes infinitely many values. For instance, the Trunk-Space type in Figure 2.18 may comprise all natural numbers. On the other hand, the generator function always returns a finite set. The PRODIGY manual [Carbonell *et al.*, 1992] contains a more detailed description of infinite types.

2.4 Search control

The efficiency of problem solving depends on the search space and the order of expanding nodes of the space. The nondeterministic PRODIGY algorithm in Figure 2.32 defines the search space, but does not specify the exploration order. The algorithm has several decision points (see Figure 2.33), which require heuristics for selecting appropriate branches of the search space.

The PRODIGY architecture includes a variety of search-control mechanisms, which combine general heuristics, domain-specific experience, and advice by the human user. Some of the basic mechanisms are an integral part of the search algorithm, hard-coded into the system; however, most mechanisms are optional, and the user can enable or disable them at her discretion.

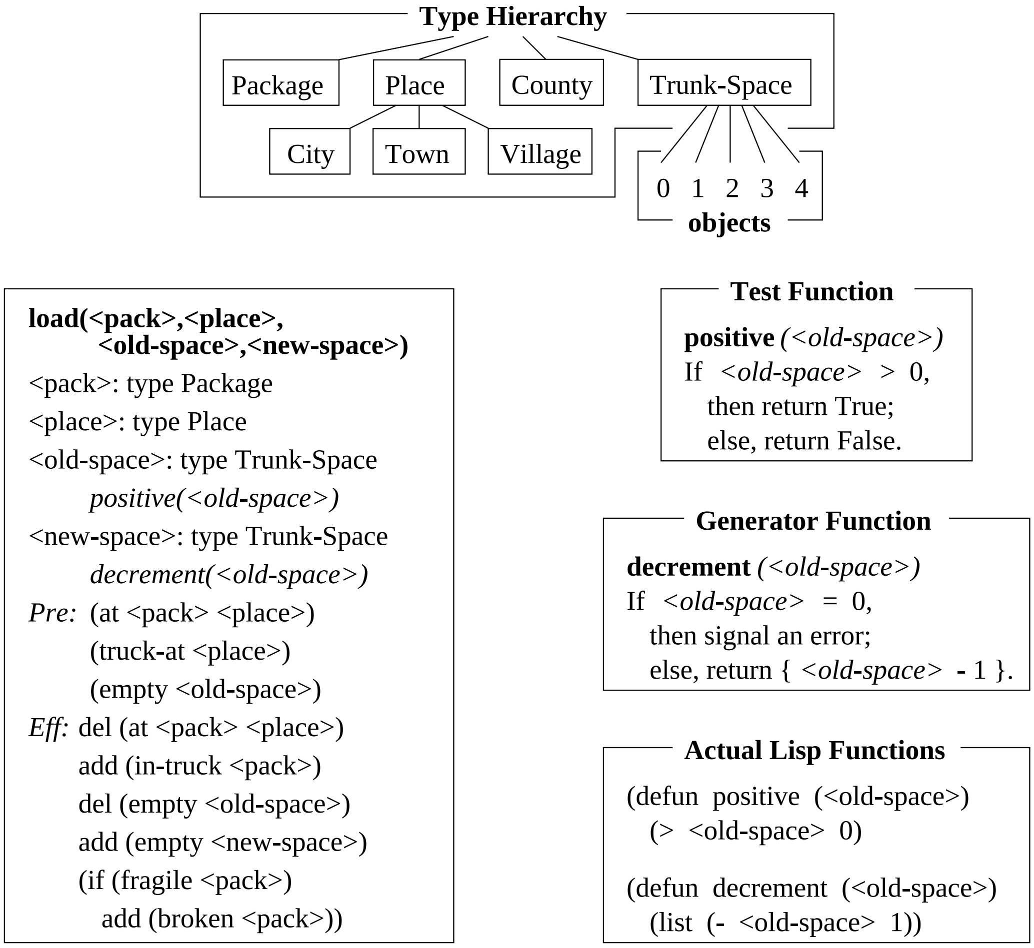


Figure 2.19: Generator function: The user provides a Lisp function, called *decrement*, which generates instances of <new-space>; these instances must belong to the specified type, Trunk-Space.

We outline some control mechanisms, including heuristics for avoiding redundant search (Section 2.4.1), main knobs for adjusting the search strategy (Section 2.4.2), and the use of control rules to guide the search (Section 2.4.3). The reader may find an overview of other control techniques in the article by Blythe and Veloso [1992], which explains dependency-directed backtracking in PRODIGY, the use of limited look-ahead, and some heuristics for choosing appropriate subgoals and instantiations.

2.4.1 Avoiding redundant search

We describe three basic techniques for eliminating redundant branches of the search space. These techniques improve the performance in almost all domains and, hence, they are hard-coded into the search algorithm, which means that the user cannot turn them off.

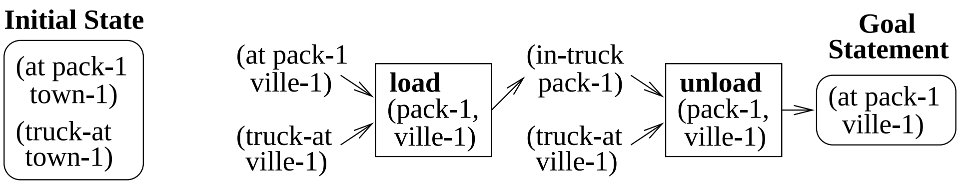


Figure 2.20: Goal loop in the tail: The precondition (at pack-1 ville-1) of the **load** operator is the same as the goal literal; hence, the solver has to backtrack and choose another operator.

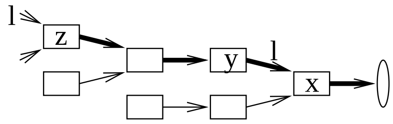


Figure 2.21: Detection of goal loops: The backward changer compares the precondition literals of a newly added operator z with the links between z and the goal statement. If some precondition l is identical to one of the link literals, then the solver backtracks.

Goal loops

We 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 2.20). The solver first adds **unload**(pack-1,ville-1) and then may try 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, hence, achieving it is as difficult as solving the original problem.

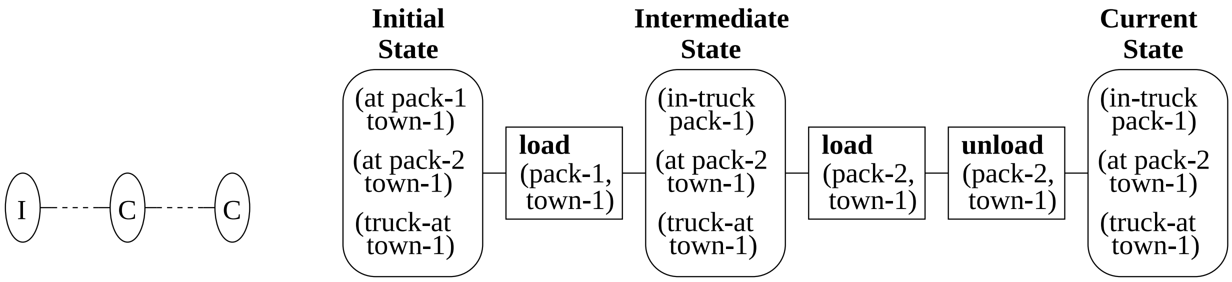
We call it a *goal loop,* which arises when a precondition of a newly added operator is identical to the literal of some link on the path from this operator to the goal statement. We illustrate it in Figure 2.21, where thick links mark the path from a new operator z to the goal. The precondition l of z makes a loop with an identical precondition of x, achieved by y.

When the problem solver adds an operator to the tail, it compares the operator's preconditions with the links between this operator and the goal. If the solver detects a goal loop, it backtracks and tries either a different instantiation of the operator or an alternative operator that achieves the same subgoal. For example, the solver may generate a new instantiation of the **load** operator, **load**(pack-1,town-1).

State loops

The problem solver also watches for loops in the head of an incomplete solution, 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 2.22a), then the solver discards the current incomplete solution and backtracks.

We illustrate a state loop in Figure 2.22, where the application of two opposite operators, **load** and **unload**, leads to a repetition of an intermediate state. The solver would detect this redundancy and either delay the application of **unload** or use a different instantiation.



(a) State loop. (b) Example of a loop.

Figure 2.22: State loops in the head of an incomplete solution: If the current state C is the same as one of the previous states, then the problem solver backtracks. For example, if PRODIGY applies **unload**(pack-2,town-1) immediately after **load**(pack-2,town-1), then it creates a state loop.

Satisfied links

Next, we describe the detection of redundant tail operators, illustrated in Figure 2.23. In this example, PRODIGY is solving the problem in Figure 2.2, and it has constructed the tail shown in Figure 2.23(a). Note that the literal (truck-in ville-1) is a precondition of two different operators in this solution, **unload**(pack-1,ville-1) and **unload**(pack-2,ville-1). Thus, they introduce two identical subgoals, and the solver adds two copies of the operator **leave-town**(town-1,ville-1) to achieve these subgoals.

Such situations arise because PRODIGY links each tail operator to only one subgoal, which simplifies the maintenance of links. When the solver applies an operator, it detects and skips redundant parts of the tail. For example, suppose that it has applied the two **load** operators and one **leave-town**, as shown in Figure 2.23(b). The precondition (truck-in ville-1) of the tail operator **unload** now holds in the current state, and the solver skips the tail operator **leave-town**, linked to this precondition.

When a tail operator achieves a precondition that holds in the current state, we call the corresponding link *satisfied.* We show this situation in Figure 2.24(a), where the precondition l of x is satisfied, which makes the dashed operators redundant.

The problem solver keeps track of satisfied links, and updates their list after each modification of the current state. When the solver selects a tail operator to apply (line 1b in Figure 2.8) or a subgoal to achieve (line 1c), it ignores the tail branches that support satisfied links. Thus, it would not consider the dashed operators in Figure 2.24 and their preconditions.

If the algorithm applies the operator x, it discards the dashed branch that supports a precondition of x (Figure 2.24b). This strategy allows the deletion of redundant operators from the tail. Note that the solver 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.

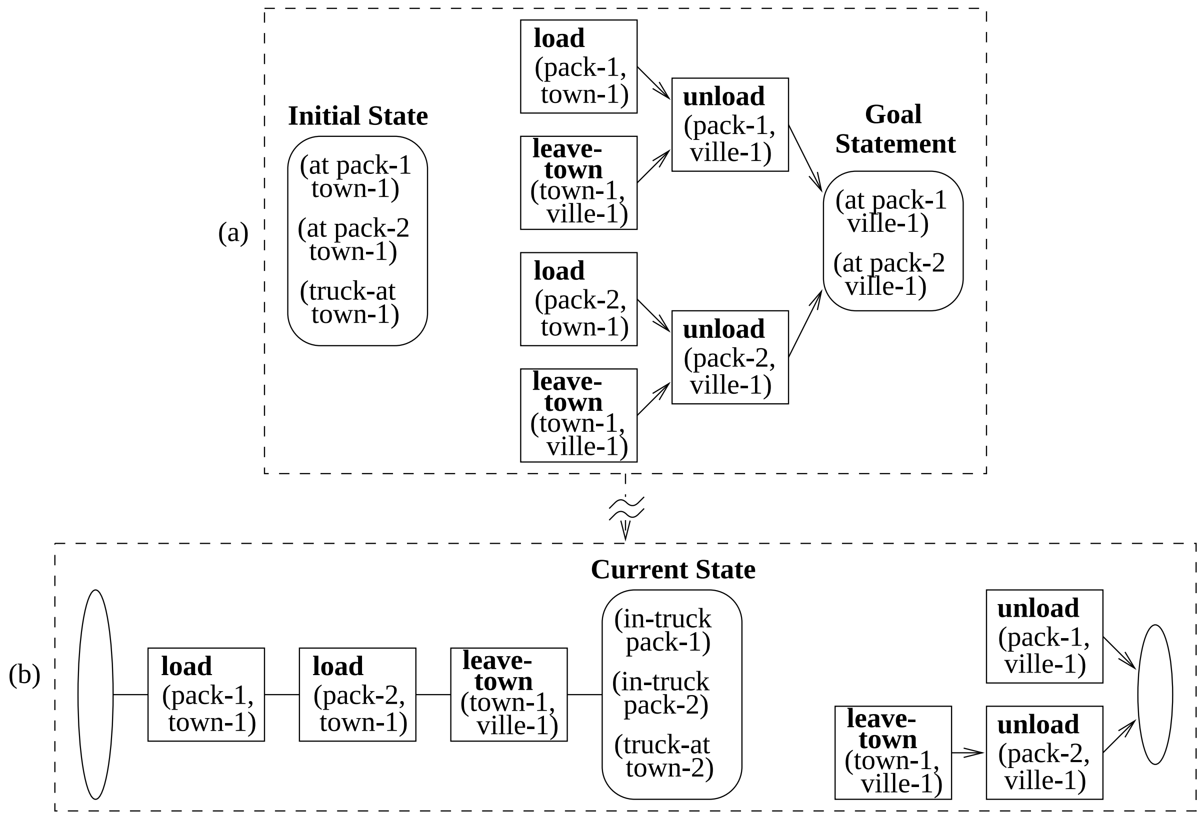


Figure 2.23: Satisfied link: After the solver has applied three operators, it notices that all preconditions of **unload**(pack-2,ville-1) hold in the current state; hence, it omits the tail operator **leave-town**, which is linked to a satisfied precondition of **unload**.

2.4.2 Knob values

The PRODIGY architecture includes several knob variables, which allow the user to adjust the search strategy to a current domain. and changes in their values may have a drastic impact on performance. Some of the knobs are numerical values, such as the search depth, whereas others specify the choices among alternative techniques and heuristics. We list some of the main knob variables, which were used in our experiments with the $\mathcal{S}$HAPER system.

Depth limit

The user usually limits the search depth, which results in backtracking upon reaching the pre-set limit. If the system explores all branches of the search space to the specified depth and does not find a solution, then it terminates with failure. 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.

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

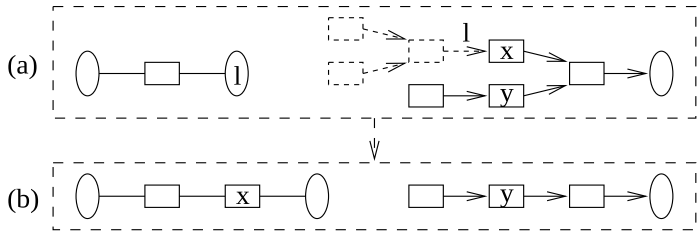


Figure 2.24: Identification of satisfied links: The solver keeps track of all link literals that are satisfied in the current state, and disregards the tail operators that support these satisfied literals.

The effect of these bounds varies across domains and specific problems. Sometimes, they improve not only solution quality but also efficiency, by preventing a long descent into a branch that has no solutions. In other domains, they cause an extensive search instead of fast generation of a suboptimal solution. If the search space has no solution within the specified bound, then the system fails to solve the problem, which means that a depth limit may cause a failure on a solvable problem.

Time limit

By default, the problem solver runs until it either finds a solution or exhausts the available search space. If it takes too long, the user may enter a keyboard interrupt or terminate the execution.

Alternatively, she may pre-set a time limit before invoking the solver and then the system automatically interrupts the search upon reaching this limit. We will analyze the role of time limits in Chapter 7.

The user may also bound the number of expanded nodes in the search space, which causes an interrupt upon reaching the specified node number. If she limits both running time and node number, then the search terminates after hitting either bound.

Search strategies

The system normally uses depth-first search and terminates upon finding any complete 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 request *all* solutions to a given problem. Then, the solver explores the entire search space and outputs all available solutions, until it exhausts the space, gets a keyboard interrupt, or reaches a time or node bound. 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 solutions. If the solver gets an interrupt, it outputs the best solution found by that time.

(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> if 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 2.25: Examples of control rules, which encode domain-specific heuristics for guiding PRODIGY search. The user may provide rules that represent her knowledge about the domain. Moreover, the system includes several mechanism for automatic construction of control heuristics.

2.4.3 Control rules

The efficiency of depth-first search crucially depends on the heuristics for selecting appropriate branches of the search space, as well as on the order of exploring these branches. The PRODIGY architecture 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 general domain-independent techniques.

A *control rule* is an *if-then* rule that specifies appropriate branching decisions, which may depend on the current state, subgoals, and other features of the current incomplete solution, as well as on the global state of the search space. The PRODIGY domain 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 development of these mechanisms has been one of the main goals of the PRODIGY project.

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 incomplete solution, the rule generates one or more promising choices. For example, consider the control rule in Figure 2.25(a). When the problem solver 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* determines inappropriate choices and removes them from the search space. For instance, the rule in Figure 2.25(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 2.25 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, but 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 system 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 incomplete solution matches several select rules, the system arbitrarily selects one of them. If no select rules are applicable, then all available branches become candidates. Next, PRODIGY applies all reject rules that match the current solution 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; hence, they may prevent the system from solving some problems.

After using select and reject rules to prune branches at the current decision point, PRODIGY applies prefer rules to determine the order of exploring the remaining branches. If the system has no applicable prefer rules, or the applicable rules contradict each other, then it relies on general heuristics for selecting 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] has implemented several techniques that improve the matching efficiency; however, the study of the trade-off between matching time and search reduction remains an open problem.

2.5 Completeness

A search algorithm 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 for some problems.

Even though researchers used the PRODIGY search engine in multiple studies of learning and search, the question of its completeness had remained unanswered for several years. Veloso demonstrated the incompleteness of PRODIGY4 in 1995. During the work on $\mathcal{S}$HAPER, we further investigated completeness issues, in collaboration with Blythe.

The investigation showed that, to date, all PRODIGY algorithms had been incomplete; moreover, it revealed the specific reasons for their incompleteness. Then, Blythe implemented a complete solver by extending the PRODIGY4 search engine. We compared it experimentally with the incomplete system, and demonstrated that the extended algorithm is almost as efficient as PRODIGY and solves a wider range of problems [Fink and Blythe, 1998]. We now report the results of this work on completeness.

We have already shown that PRODIGY1 and PRODIGY2 do not interleave goals and sometimes fail to solve very simple problems. NOLIMIT, PRODIGY4, and FLECS use a more flexible strategy, and they fail 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 PRODIGY is not a major handicap. Since the search space of most problems is very large, a complete exploration of the space is not feasible, which makes any problem solver "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 bonuses. First, we ensure that the system 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 solver is complete, it does not rule out any solutions and is able to find such a simple solution early in the search.

The incompleteness of means-ends analysis in PRODIGY comes from two sources. First, the problem solver 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 solver fails because it did not take measures earlier. Second, PRODIGY ignores the conditions of if-effects that do not achieve any subgoal. Sometimes, such effects negate goals or preconditions of other operators, which may cause a failure.

We achieve completeness by adding crucial 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 use of the information learned in failed old branches, and give an extended search algorithm (Sections 2.5.1 and 2.5.2). We believe that this method will prove useful for developing complete versions of other search algorithms.

The extended domain language of PRODIGY has two features that aggravate the completeness problem (Section 2.5.3), which are not addressed in the extended algorithm. 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 problems are undecidable.

We prove that the extended algorithm is complete for domains that have no eager inference rules and functional types (Section 2.5.4). Then, in Section 2.5.5, we give experimental results on the relative performance of PRODIGY4 and the extended solver. We conclude with the summary and discussion of the main results (Section 2.5.6).

2.5.1 Limitation of PRODIGY means-ends analysis

GPS, PRODIGY1, and PRODIGY2 were not complete because they did not explore all branches in their search space. The incompleteness of later algorithms has a deeper reason: they do not try to achieve tail 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 (see Figure 2.26a). The precondition (truck-at town-1) of **unload** is true in the initial state. The problem solver 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 2.26(a).

Now all preconditions are satisfied, and the solver's only choice is to apply **leave-town** (Figure 2.26b). The application leads straight into an inescapable trap, where the truck is stranded in ville-1 without a supply of extra fuel. The algorithm may backtrack and consider different instantiations of **load**, but they will eventually lead to the same trap.

To avoid such traps, a solver must sometimes add operators for achieving literals that are true in the current state and have not been linked with any tail operators. Such literals are called *anycase subgoals*. The challenge is to identify anycase subgoals among the preconditions of tail 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 their FLECS search algorithm; however, it proved to cause 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 precondi-

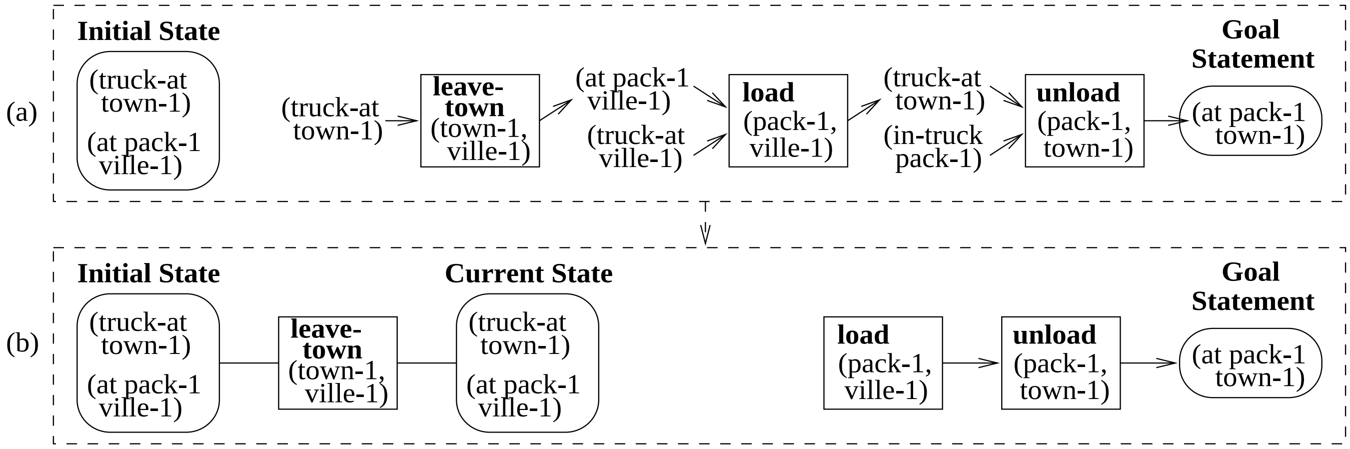


Figure 2.26: Incompleteness of means-ends analysis in the PRODIGY system: The solver does *not* consider fueling the truck before the application of **leave-town**(town-1,ville-1). Since the truck cannot leave ville-1 without extra fuel, PRODIGY fails to find a solution.

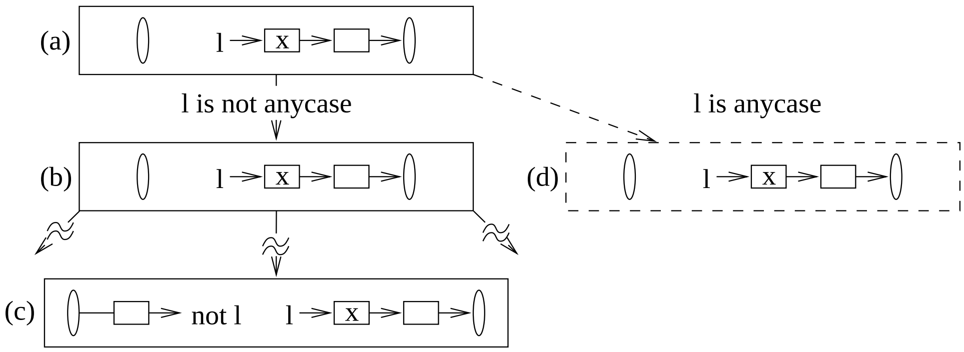


Figure 2.27: Identifying an anycase subgoal: When PRODIGY adds a new operator x (a), the preconditions of x are *not* anycase subgoals (b). If some application negates a precondition l of x (c), the solver marks l as anycase and expands the corresponding new branch of the search space (d).

tions that are true in the current state, if the preconditions are not explicitly linked to the corresponding literals of the state. Even though this approach is more efficient than viewing all preconditions as anycase subgoals, it considerably increases branching and often makes search impractically slow.

A more effective solution is based on the *use of information learned in failed branches of the search space.* Let us look again at Figure 2.26. The problem solver 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. The solver 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 is later negated by an operator application. We use this observation to identify anycase subgoals: *a precondition or a goal literal is an anycase subgoal if, at some point of the search, an application negates it.*

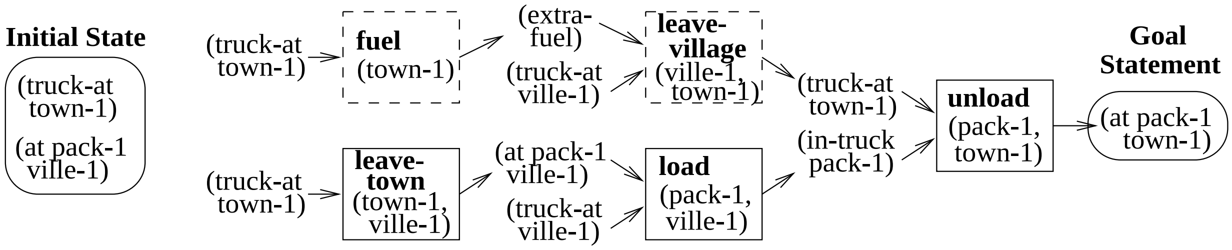


Figure 2.28: Achieving the subgoal (track-at town-1), which is satisfied in the current state. First, the solver constructs a three-operator tail, shown by solid rectangles. Then, it applies the **leave-town** operator and marks the precondition (truck-at town-1) of **unload** as an anycase subgoal. Finally, PRODIGY backtracks and adds the two dashed operators, which achieve this subgoal.

In Figure 2.27, we illustrate a technique for identifying anycase subgoals in Figure 2.27. Suppose that the problem solver adds an operator x, with a precondition l, to the tail (node a in the picture). The solver creates the branch where l is not an anycase subgoal (node b). If, at some descendent, an application of some operator negates l and if it was true before the application, then l is marked as anycase (node c). If the solver fails to find a solution in this branch, it eventually backtracks to node a. If l is marked as anycase, the problem solver creates a new branch, where l is an anycase subgoal (node d).

If several preconditions of x are marked as anycase, the solver creates the branch where they all are anycase subgoals. Note that, during the exploration of this new branch, the algorithm may mark some other preconditions of x as anycase. If it again backtracks to node a, then it creates a branch where the newly marked preconditions are also anycase subgoals.

Let us see how this mechanism works for the example problem. The solver first assumes that the preconditions of **unload**(pack-1,town-1) are not anycase subgoals. It builds the tail shown in Figure 2.26 and applies **leave-town**, negating the precondition (truck-at town-1) of **unload**. The solver then marks this precondition as anycase.

Eventually, the algorithm backtracks, creates the branch where (truck-at town-1) is an anycase subgoal, and uses the operator **leave-village**(ville-1,town-1) to achieve this subgoal (see Figure 2.28). The problem solver then constructs the tail shown in Figure 2.28, 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)" (note that the precondition (truck-at ville-1) of **leave-village** is satisfied after applying **leave-town**).

When the solver identifies the set of all satisfied links (Section 2.4.1), it does not includes anycase links into this set; hence, it never ignores the tail operators that support anycase links. For example, consider the tail in Figure 2.28: the anycase precondition (truck-at town-1) of **unload** holds in the state, but the solver does *not* ignore the operators that support it.

We also have to modify the detection of goal loops, described in Section 2.4.1. For instance, consider again Figure 2.28: the precondition (truck-at town-1) of **fuel** makes a loop with the identical precondition of **unload**; however, the solver should *not* backtrack.

Since this precondition of **unload** is an anycase subgoal, it must not cause goal-loop backtracking. We use Figure 2.21 to generalize this rule: if the precondition l of x is an anycase subgoal, then the identical precondition of z does *not* make a goal loop.

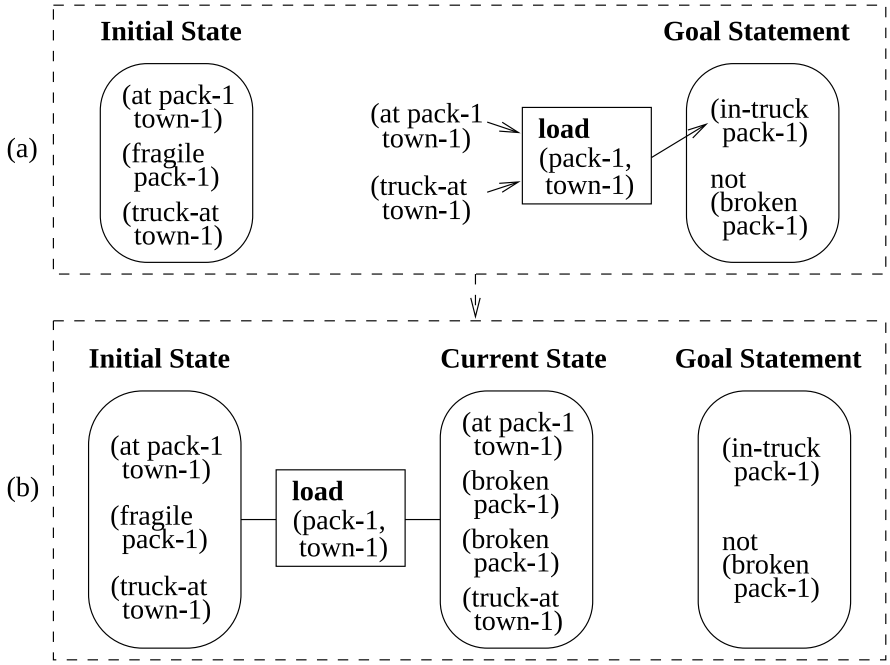


Figure 2.29: Failure because of a clobber effect: The application of the **load** operator results in the breakage of the package, and no further actions can undo this damage.

2.5.2 Clobbers among if-effects

In Figure 2.29, we illustrate another source of incompleteness: the use of if-effects. The goal is to load fragile pack-1, without breaking it. The problem solver 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 2.29a), and the solver's only choice is to apply **load**. The application causes the breakage of pack-1 (Figure 2.29b), and no further search improves the situation. The solver may try other instances 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 solver backtracks and tries to find another instance of **load**, or another operator, which 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 another alternative: negate its conditions [Pednault, 1988a; Pednault, 1988b]. It 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 to add a new decision point, where the algorithm determines whether it should negate a clobber's conditions.

Introducing this new decision point for every if-effect will ensure completeness, but 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. *An 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 literal of the goal statement, an operator precondition, or a condition of

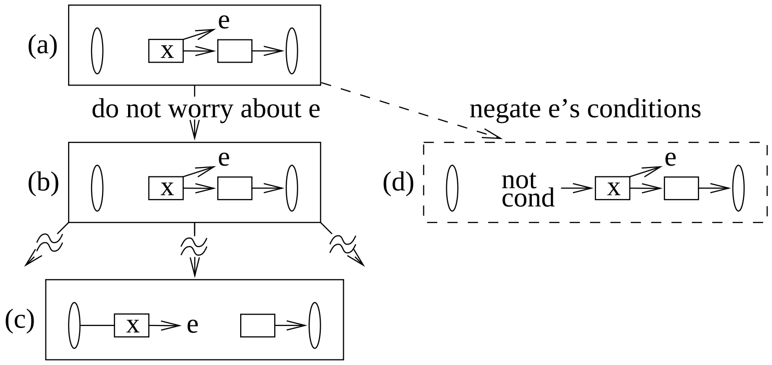


Figure 2.30: Identifying a clobber effect. When the solver adds a new operator x (a), it does *not* try to negate the conditions of x's if-effects (b). When applying x, PRODIGY checks whether if-effects negate any subgoals that were true before the application (c). If an if-effect e of x negates some subgoal, the solver marks e as a clobber and adds the respective branch to the search space (d).

an if-effect that achieves another subgoal. Thus, we again use information learned in failed branches to guide the search.

We illustrate the mechanism for identifying clobbers in Figure 2.30. Suppose that the problem solver adds an operator x with an if-effect e to the tail, and that this operator is added for the sake of some other of its effects (node a in Figure 2.30); that is, e is not linked to a subgoal. Initially, the problem solver does not try to negate e's conditions (node b). If, at some descendent, x is applied and its effect e negates a subgoal that was true before the application, then the solver marks e as a potential clobber (node c). If the solver fails to find a solution in this branch, it backtracks to node a. If e is now marked as a clobber, the solver adds the negation of e's conditions, *cond*, to the operator's preconditions (node d). If an operator has several if-effects, the solver uses a separate decision point for each of them.

In the example of Figure 2.29, the application of **load**(pack-1,town-1) negates the goal "not (broken pack-1)" and the problem solver marks the if-effect of **load** as a potential clobber (see Figure 2.31). Upon backtracking, the solver adds the *negation* of the clobber's condition (fragile pack-1) to the preconditions of **load**. The solver uses **cushion** to achieve this new precondition and generates the solution "**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 2.32. We show its main decision points in Figure 2.33, where thick lines mark the points absent in PRODIGY.

2.5.3 Other violations of completeness

The PRODIGY system has two other sources of incompleteness, which arise from the advanced features of the domain language. We have not addressed them in our work; hence, the use of these language features may violate the completeness of the extended algorithm.

[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*.

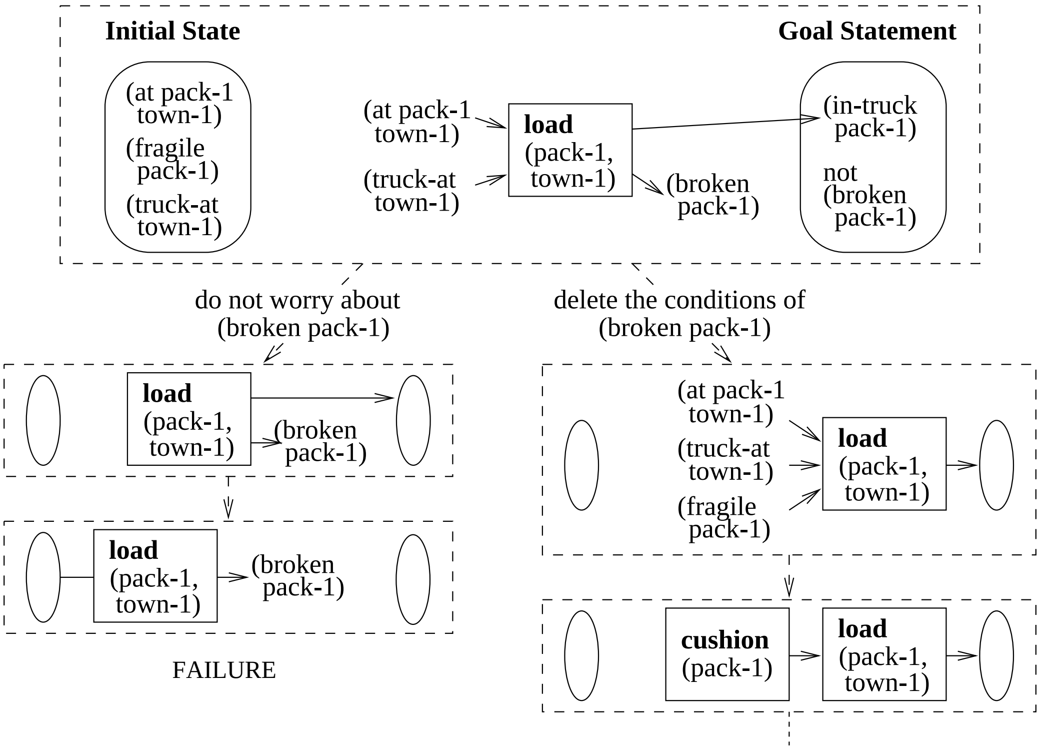


Figure 2.31: Negating a clobber. When **load** is applied, its if-effect clobbers one of the goal literals. The solver backtracks and adds the **cushion** operator, which negates the conditions of this if-effect.

Eager inference rules

If a domain includes eager rules, the system may fail to solve simple problems. For example, consider the rules in Figure 2.14 and the problem in Figure 2.34(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 2.34(b). If we applied the operator **leave-town**(town-1,town-2), it would negate the rule's preconditions, which would cause the deletion of (truck-in county-1). In other words, the application of **leave-town** would immediately solve the problem.

The system 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 problem solver terminates with failure.

To summarize, the solver sometimes has to negate the preconditions of eager rules that have clobber effects, but it does not consider this option. We plan to implement the negation of clobber rules as in the future.

RASPUTIN-**Back-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 and establish a link from *op* 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 descendants 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 2.32: Backward-chaining procedure of the RASPUTIN problem solver; it includes new decision points (lines 6c and 7c), which ensure completeness of PRODIGY means-ends analysis.

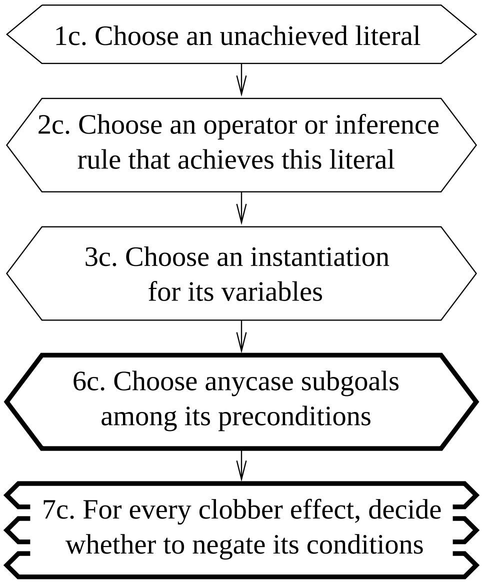
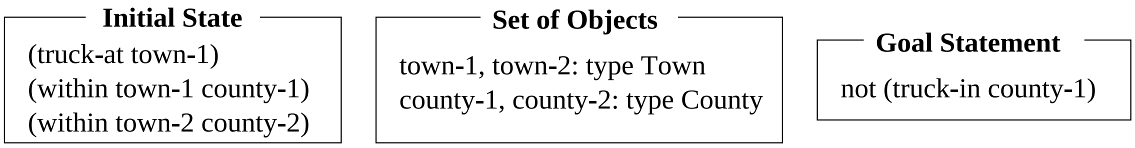
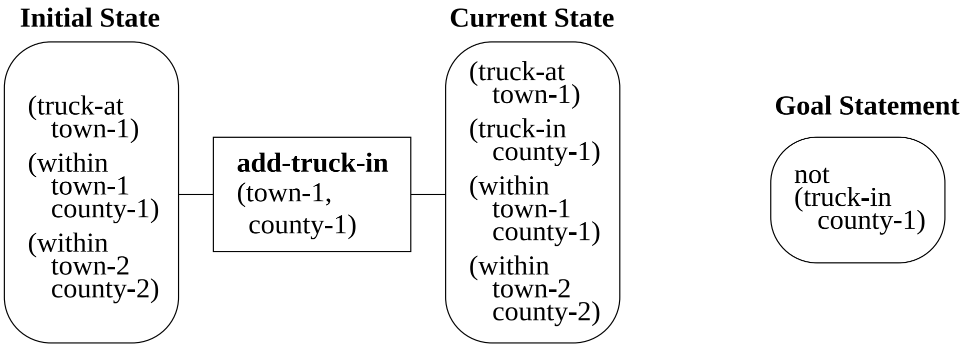


Figure 2.33: Main decision points in RASPUTIN's backward-search procedure, summarized in Figure 2.32; thick lines show the decision points that differentiate it from PRODIGY (see Figure 2.8).



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



(b) Application of an inference rule.

Figure 2.34: Failure because of an eager inference rule: The PRODIGY solver does *not* attempt to negate the preconditions of the rule **add-truck-in**(town-1,county-1), which clobbers the goal.

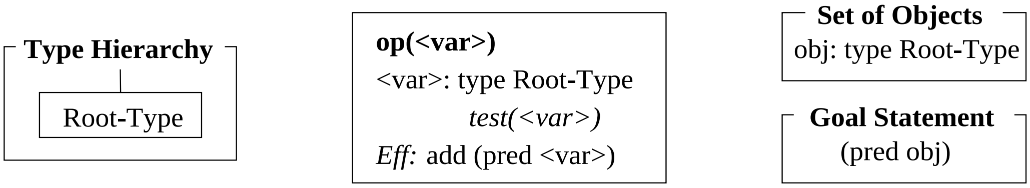


Figure 2.35: Reducing an arbitrary decision task to a PRODIGY problem; the test function is a Lisp procedure that encodes the decision task. Since functional types allow encoding of undecidable problems, they cause incompleteness of PRODIGY search.

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 2.35. If the object obj satisfies the test function, then the solver uses the operator **op**(obj) to achieve the goal; otherwise, the problem has no solution.

The test function may be any Lisp program, which has full 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 scope of classical search. A related open problem is defining restricted classes of useful functions, which do not cause computational difficulties, and ensuring completeness for these functions.