A Complete Bidirectional Planner

Eugene Fink
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
eugene@cs.cmu.edu
http://www.cs.cmu.edu/∼eugene

Jim Blythe
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
blythe@isi.edu
http://www.isi.edu/∼blythe

Abstract

The PRODIGY system is based on *bidirectional planning*, which is a combination of goal-directed backward chaining with simulation of plan execution. Experiments have demonstrated that it is an efficient technique, a fair match to other successful planning systems. The question of completeness of bidirectional planning, however, has remained unanswered.

We show that PRODIGY is not complete and discuss the advantages and drawbacks of its incompleteness. We then develop a complete bidirectional planner and compare it experimentally with PRODIGY. We demonstrate that the complete planner is almost as efficient as PRODIGY and solves a wider range of problems.

Introduction

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

Meanwhile, PRODIGY researchers extended means-ends analysis and developed a family of planners based on the combination of goal-directed backward chaining with simulation of plan execution (Veloso *et al.* 1995). We call them *bidirectional planners*, which include PRODIGY2 (Minton 1988), NOLIMIT (Veloso 1989; Veloso 1994), PRODIGY4 (Veloso *et al.* 1995), and FLECS (Veloso and Stone 1995). These planners keep track of the planning-domain state that results from executing parts of the currently constructed plan. They use the domain state to guide the goal-directed reasoning. Least commitment proved inefficient for bidirectional search, and Veloso developed *casual commitment*, based on instantiating all variables as early as possible (Veloso 1994).

Kambhampati and Srivastava developed a framework that generalizes both least-commitment and bidirectional search, as well as direct forward search. They implemented Universal Classical Planner (UCP), which can use all these search strategies (Kambhampati and Srivastava 1996a; Kambhampati and Srivastava 1996b).

Blum and Furst have recently developed GRAPHPLAN (Blum and Furst 1995), which uses the domain state in a different way. They implemented propagation of constraints from the initial state of the domain, which enables their planner to identify some operators whose preconditions cannot be satisfied. The planner then discards these operators and uses backward chaining to construct the plan from the remaining operators. GRAPHPLAN performs the forward constraint propagation prior to the search for a solution plan. Unlike PRODIGY, it does *not* use forward search from the initial state.

Experiments have demonstrated that bidirectional search is an efficient technique, a fair match to other modern planners (Stone *et al.* 1994), and that PRODIGY and backward chainers perform well in different domains. Some problems are more suitable for casual-commitment bidirectional search (Veloso and Blythe 1994), whereas others require backward chaining (Barrett and Weld 1994).

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

To date, all bidirectional planners have been incomplete. PRODIGY2 does not interleave goals and sometimes fails to solve very simple problems. NOLIMIT, PRODIGY4, and FLECS use a more flexible strategy, and their incompleteness arises less frequently. Veloso and Stone proved the completeness of FLECS using simplifying assumptions (Veloso and Stone 1995), 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 planner "practically" incomplete. If incompleteness comes up only in a fraction of problems, it is a fair payment for efficiency.

If we achieve completeness *without compromising efficiency*, we get two bonuses. First, we ensure that the planner solves every problem whose search space is sufficiently small for complete exploration. Second, incompleteness may occasionally rule out a simple solution to a large-scale problem,

Copyright 1998, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

causing an extensive search instead of an easy win. If a planner is complete, it does not rule out any solutions and is able to find such a simple solution early in the search.

We identify the two specific reasons for the incompleteness of means-ends analysis and develop the first complete bidirectional planner, by extending the PRODIGY search algorithm. The planner supports a rich domain language, which allows the use of conditional effects and complex precondition expressions (Carbonell *et al.* 1992).

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. We believe that this method will prove useful for developing complete versions of other search algorithms.

We outline a proof of completeness and then demonstrate experimentally that the new planner is almost as efficient as PRODIGY and that it solves more problems.

Description of Planning Problems

We define a *planning domain* by a set of object types and a library of operators that act on objects of these types. An operator is defined by its *preconditions* and *effects*. The preconditions of an operator are the conditions that must be satisfied before the execution. They are represented by a logical expression with negations, conjunctions, disjunctions, and universal and existential quantifiers. The effects are the list of predicates added to or deleted from the current state of the domain upon the execution.

We may specify conditional effects, whose outcome depends on the domain state. A conditional effect is defined by *conditions* and *actions*. If the conditions hold, the effect changes the state, according to its actions. Otherwise, it does not affect the state.

In Figure 1, we give an example of a simple domain. The domain has two types of objects, Package and Place. The Place type includes two subtypes, Town and Village.

A truck carries packages between towns and villages. The truck's tank of fuel is sufficient for only one ride. Towns have gas stations, so the truck can refuel before leaving a town. Villages do not have gas stations; if the truck comes to a village without a supply of extra fuel, it cannot leave. The truck can get extra fuel supply 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 *planning problem* is defined by a list of objects, an *initial state*, and a *goal statement*. The initial state is a set of literals. The goal statement is a condition that must hold after executing a plan. A *solution* is a sequence of operators that can be executed from the initial state to achieve the goal. We give an example of a problem in Figure 2. The task in this problem is to deliver two packages from town-1 to town-2. We may solve it by the following plan: "**load**(pack-1, town-1), **load**(pack-2, town-1), **leave-town**(town-1, town-2), **unload**(pack-1, town-2), **unload**(pack-2, town-2)."

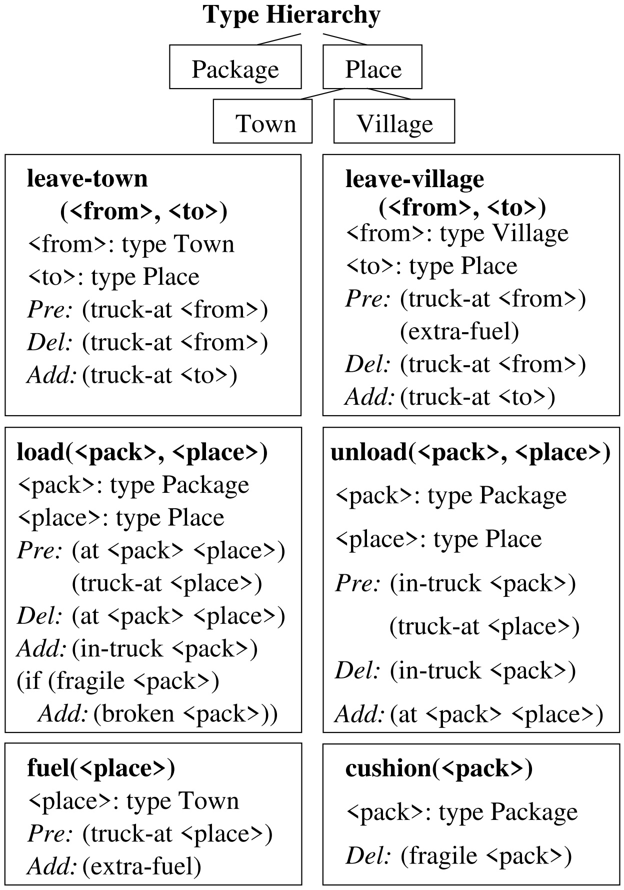

Figure 1: Trucking Domain.

Foundations of Bidirectional Planning

We now give basics of bidirectional planning (Veloso *et al.* 1995). We omit the methods for handling disjunctive and quantified preconditions. All PRODIGY planners 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.

Given a problem, most planning systems begin with the empty plan and modify it until a solution is found. Examples of modifications include adding an operator, instantiating or constraining a variable in an operator, and imposing an ordering constraint.

The PRODIGY planner also solves a problem by adding operators and constraints to the initial empty plan. In PRODIGY, a plan consists of two parts, a total-order *head-plan* and tree-structured *tail-plan* (see Figure 3). The root of the tail-plan's tree is the goal statement G, the other nodes are operators, and the edges are ordering constraints. The tail-plan is built by a backward chainer, which starts from the goal statement and adds operators, one by one, to achieve goal literals and preconditions of tail-plan operators. The planner completely instantiates an operator when adding it to the tail-plan. The head-plan is a sequence of instantiated operators that can be executed from the initial state. The state C achieved by executing the head-plan is the *current state*. In Figure 4, we give

Set of Objects
pack-1, pack-2: type Package
town-1, town-2: type Town
ville: type Village

Initial State
(at pack-1 town-1)
(at pack-2 town-1)
(truck-at town-1)

Goal Statement
(at pack-1 town-2)
(at pack-2 town-2)

Figure 2: Problem in the Trucking Domain.

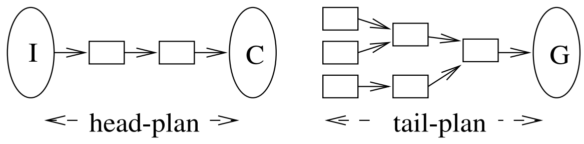

Figure 3: Representation of a plan.

an example of an incomplete plan, constructed by PRODIGY during its search for a solution.

Given an initial state I and a goal statement G, PRODIGY begins with the empty plan and modifies it, step by step, until it builds a solution. The head and tail of the initial plan are empty, and the current state is the same as initial, $C = I$.

At each step, PRODIGY can modify the current plan in one of two ways. First, it can add an operator to the tail-plan, to achieve a goal literal or a precondition of another operator. The planner establishes a link from the newly added operator to the literal it achieves and the corresponding ordering constraint. If the planner uses a conditional 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 with any operator of the tail-plan.* Unsatisfied goal literals and preconditions are called *subgoals*.

Second, PRODIGY can move some operator *op* from the tail to the head. The preconditions of *op* must be satisfied in the current state C. The operator *op* becomes the last operator of the head, and the current state is updated to account for the effects of *op*. Moving an operator to the head is called the *application* of the operator.

In Figure 5, we summarize the PRODIGY search algorithm, which explores the space of possible plans. The algorithm

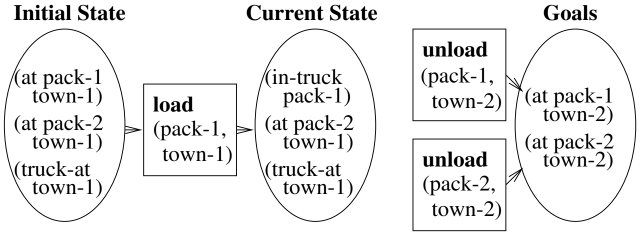

Figure 4: Example of an incomplete plan.

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

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

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

Figure 5: The PRODIGY search algorithm.

includes five decision points, which give rise to different branches of the search space. It can backtrack over the choice of a subgoal (Line 1b), an operator that achieves it (Line 2b), and the operator's instantiation (Line 4b). It also backtracks over the decision to apply an operator (Line 2a) and the choice of an "applicable" tail-plan operator (Line 1c). The two latter choices enable the planner to consider different ordering of operators in the head-plan. These two choices are essential for completeness; they are analogous to the choice of ordering constraints in least-commitment planners.

The search terminates when the head-plan achieves the goals; that is, the goal statement G is satisfied in C. If the tail-plan is not empty at that point, it is dropped.

The incompleteness comes from two sources. First, the planner does not add operators for achieving literals that are true in the current state. Intuitively, the planner ignores potential troubles until they actually arise. Sometimes, it is too late and the planner fails because it did not take measures earlier. Second, PRODIGY ignores the conditions of conditional effects that do not establish any subgoal. Sometimes, such effects negate goals or preconditions of other operators, which may cause a failure. We discuss these two situations and show how to extend the search to make it complete.

Limitation of Means-Ends Analysis

GPS and PRODIGY2 were not complete because they did not explore all branches in their search space. The incompleteness of later means-end analysis planners has a deeper rea-

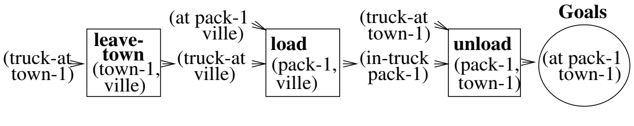

Figure 6: Incompleteness of means-end analysis.

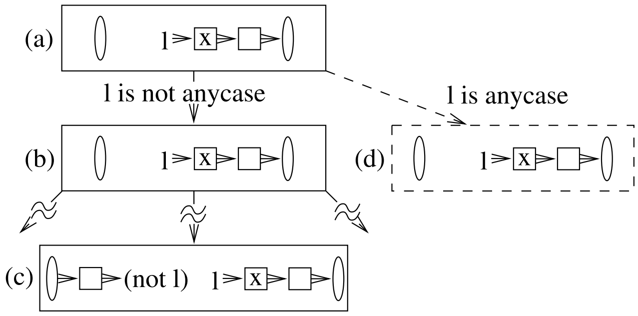

Figure 7: Identifying an anycase subgoal.

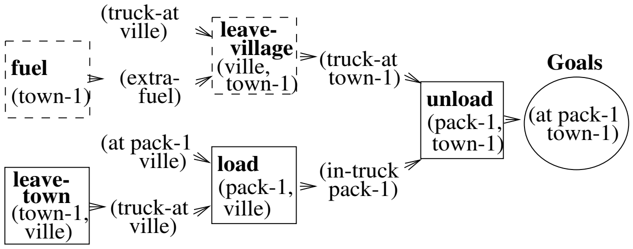

Figure 8: Subgoaling on literals true in the current state.

son: they do not try to achieve tail-plan preconditions that hold in the current state.

For example, suppose that the truck is in town-1, pack-1 is in ville, 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 6). The precondition (truck-at town-1) of **unload** is true in the initial state. The planner may achieve the other precondition, (in-truck pack-1), by adding **load**(pack-1, ville). The precondition (at pack-1 ville) of **load** is true in the initial state, and the other precondition is achieved by **leave-town**(town-1, ville) (Figure 6).

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

To avoid such traps, a planner must sometimes add operators for achieving literals that are true in the current state and have not been linked with any tail-plan operators. Such literals are called *anycase subgoals*. The challenge is to identify anycase subgoals among the preconditions of tail-plan operators.

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

Kambhampati and Srivastava used a similar approach to ensure the completeness of the Universal Classical Planner (Kambhampati and Srivastava 1996b). Their planner may add operators for achieving preconditions 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 bidirectional 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 6. The planner fails because it does not add any operator to achieve the precondition (truck-at town-1) of **unload**, which is true in the initial state. The planner 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 top-level goal is an anycase subgoal if, at some point of the search, an application negates it.*

We illustrate the identification of anycase subgoals in Figure 7. Suppose that the planner adds an operator x, with a precondition l, to the tail (plan a in Figure 7). The planner creates the branch where l is not an anycase subgoal (plan 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 (plan c). If the planner fails to find a solution in this branch, it eventually backtracks to plan a. If l is marked as anycase, the planner creates a new branch, where l is an anycase subgoal (plan d). If several preconditions of x are marked as anycase, the planner creates the branch where they all are anycase subgoals.

Let us see how this mechanism works in the example of Figure 6. The planner first assumes that the preconditions of **unload**(pack-1, town-1) are not anycase subgoals. It builds the tail-plan shown in Figure 6 and applies **leave-town**, negating the precondition (truck-at town-1) of **unload**. The planner then marks this precondition as anycase.

Eventually, the planner backtracks and creates the branch where (truck-at town-1) is an anycase subgoal. It is achieved by adding **leave-village**(ville, town-1) (see Figure 8). The planner then constructs the tail-plan shown in Figure 8, which leads to the solution (the precondition (truck-at ville) of **leave-village** is satisfied after applying **leave-town**).

Clobbers Among Conditional Effects

We illustrate the other source of incompleteness—the use of conditional effects—in Figure 9. The goal is to load fragile pack-1, without breaking it. The planner adds **load**(pack-1, town-1) to achieve (in-truck pack-1). The preconditions of **load** and the goal (not (broken pack-1)) hold in the current state, and the planner's only choice is to apply **load**. The application causes the breakage of pack-1, and no further

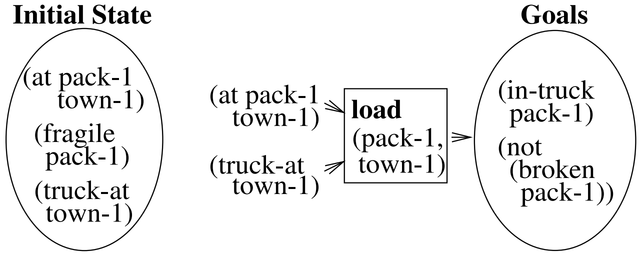

Figure 9: Failure because of a clobber effect.

planning improves the situation. The planner 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 planner backtracks and tries to find another instance of **load**, or another operator, that does not cause clobbering. If the clobber effect is not conditional, backtracking is the only way to remedy the situation.

If the clobber is a conditional effect, we should try another alternative: negate the clobber's conditions (Pednault 1988). 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 a decision point on whether to negate a clobber's conditions.

Introducing this new decision point for every conditional effect will ensure completeness, but may considerably increase branching. We avoid this problem by identifying potential clobbers among conditional 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 top-level goal, an operator precondition, or a condition of a conditional effect that achieves another subgoal. We thus again use information learned in failed branches to guide the search.

We illustrate the mechanism for identifying clobbers in Figure 10. Suppose that the planner adds an operator x, with a conditional effect e, to the tail-plan, and that this operator is added for the sake of some other of its effects (plan a in Figure 10). That is, e is not linked to a subgoal. Initially, the planner does not try to negate e's conditions (plan b). If, at some descendent, x is applied and its effect e negates a subgoal that was true before the application, then the planner marks e as a potential clobber (plan c). If the planner fails to find a solution in this branch, it backtracks to plan a. If e is now marked as a potential clobber, the planner adds the negation of e's conditions, *cond*, to the operator's preconditions (plan d). If an operator has several conditional effects, the planner uses a separate decision point for each of them.

In the example of Figure 9, the application of **load**(pack-1, town-1) negates the goal (not (broken pack-1)) and the planner marks the conditional effect of **load** as a potential clobber. Upon backtracking, the planner adds the *negation* of the clobber's condition (fragile pack-1) to the preconditions of **load**. The planner uses **cushion** to achieve this new precondition and generates the plan "**cushion**(pack-1), **load**(pack-1, town-1)."

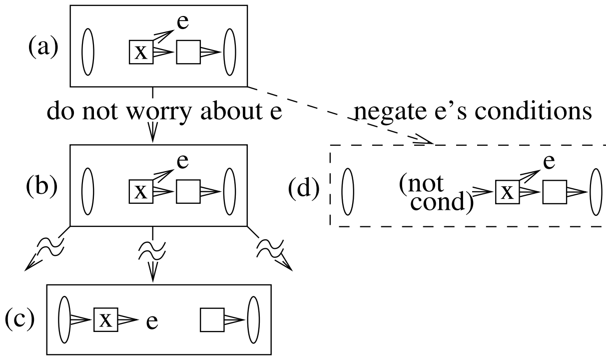

Figure 10: Identifying a clobber effect.

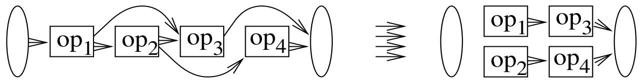

Figure 11: Converting a solution into a tail-plan.

Completeness Proof

We give a sketch of the completeness proof. We show that, *for every solvable planning problem, some sequence of choices in the planner's decision points leads to a solution.*

Suppose that a problem has a (fully instantiated) solution "$op_1, op_2, op_3, \ldots, op_n$" and that no other solution has fewer operators. We begin by defining clobber effects, subgoals, and justified effects in the solution plan.

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

A *subgoal* is a goal literal or precondition such that either (1) it does not hold in the initial state *or* (2) it is negated by some prior operator. For example, a precondition of op_3 is a subgoal if either it does not hold in the initial state or it is negated by op_1. Every subgoal in a correct solution is achieved by some operator; for example, if op_1 negates a precondition of op_3, than op_2 must achieve it.

A *justified effect* is the last effect that achieves a subgoal or negates a clobber's conditions. For example, if op_1, op_2, and op_3 all achieve some subgoal precondition of op_4, then the corresponding effect of op_3 is justified, since it is last among the three.

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

Since we consider a shortest solution, each operator in the solution plan has at least one justified effect. If we link each subgoal and each clobber's negation to the corresponding justified effect, we may use the resulting links to convert the solution into a tree-structured tail-plan. We illustrate this conversion in Figure 11. If an operator is linked to several subgoals, we use only one of the links in the tail-plan.

We now show that our algorithm can construct this tail-plan. If no subgoal holds in the initial state and the plan has no clobber effects, then the tail construction is straightforward. The nondeterministic algorithm in Figure 5 creates the desired tail-plan by always calling *Backward-Chainer* in Line 2a, choosing subgoals that correspond to the links of the desired plan in Line 1b, and selecting the appropriate operators (Line 2b) and their instantiations (Line 4b).

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

Eventually, the algorithm constructs the plan consisting of the desired tail and no head. It may then produce the solution plan by always deciding to apply in Line 2a (see Figure 5) and selecting applicable operators in the right order.

Performance of the Complete Planner

We have implemented a complete bidirectional planner, called RASPUTIN[1]. We present experimental comparison of its efficiency with that of PRODIGY4. We then demonstrate that RASPUTIN solves more problems than PRODIGY4.

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

We ran both planners on fifty problems of various complexities. We experimented with different number of cities, vans, airplanes, and packages. We randomly generated initial locations of packages, vans, and airplanes, and destinations of packages. The results are summarized in Figure 12(a), where each plus (+) denotes a problem. The horizontal axis shows PRODIGY's running time and the vertical axis gives RASPUTIN's time on the same problems. Since PRODIGY wins on all problems, all pluses are above the diagonal. The ratio of RASPUTIN's to PRODIGY's time varies from 1.20 to 1.97; its mean is 1.45.

We ran similar tests in the PRODIGY Machine Shop Domain (Gil 1991), which also does not require negating effects' conditions or planning for anycase subgoals. The task in this domain is to construct plans for making mechanical parts with specified properties, using available machining equipment. The ratio of RASPUTIN's to PRODIGY's time in this domain is between 1.22 and 1.89, with the mean at 1.39.

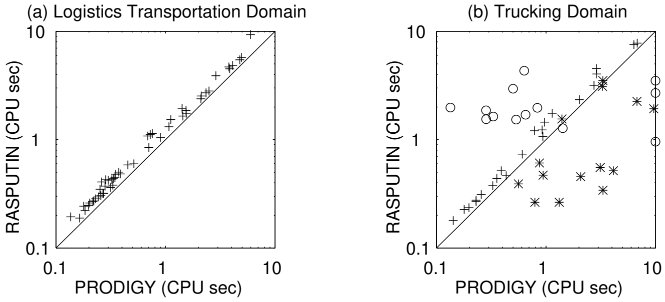

Figure 12: PRODIGY's and RASPUTIN's performance.

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

In Figure 12(b), we summarize the performance of PRODIGY and RASPUTIN on fifty problems. The twenty-two problems denoted by pluses (+) do not require the clobber negation or anycase subgoals. PRODIGY outperforms RASPUTIN on these problems, with a mean ratio of 1.27.

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

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

Conclusions

We have developed a complete bidirectional planner, by extending PRODIGY search; to our knowledge, it is the first complete planner that uses means-ends analysis. The complete planner is about 1.5 times slower than PRODIGY4 on the problems that do not require negating clobbers' conditions and planning for anycase subgoals; however, it solves problems that the incomplete planner cannot solve.

We developed the complete planner in three steps. First, we identified the specific reasons for incompleteness of previous planners. Second, we added new decision points to eliminate these reasons. Third, we implemented a search algorithm that explores the branches of the old search space first, and extends the search space only after failing to find a solution in the old space's branches. We conjecture that this

[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. Besides, his name comes from the Russian word *rasputie*, which means *decision point*.

three-step approach may prove useful for enhancing other incomplete planners.

The planner uses information from failed branches in its branching decisions, which means that it must perform depth-first search. We cannot use breadth-first or best-first search; however, breadth-first search in casual-commitment bidirectional planners is impractically slow anyways, due to a large branching factor.

Acknowledgements

PRODIGY2 was designed by Steven Minton, Jaime Carbonell, Craig Knoblock, and Dan Kuokka. The next version, NOLIMIT, was created by Manuela Veloso and Daniel Borrajo. Based on these two algorithms, Jim Blythe, Manuela Veloso, Xuemei Wang, Dan Kahn, and Alicia Pérez implemented PRODIGY4. Finally, Manuela Veloso and Peter Stone designed FLECS, the last bidirectional planner before RASPUTIN.

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

References

Anthony Barrett and Dan Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1636–1642, 1995.

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

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

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

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

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

Subbarao Kambhampati and Biplav Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 261–271. IOS Press, Amsterdam, Netherlands, 1996.

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

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

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

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

Edwin P. D. Pednault. Synthesizing Plans that Contain Actions with Context-Dependent Effects. *Computational Intelligence*, 4:356–372, 1988.

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

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

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

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

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

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

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

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

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

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

Qiang Yang, Josh Tenenberg, and Steve Woods. On the implementation and evaluation of ABTWEAK. *Computational Intelligence*, 12(2):295–318, 1996.