To Appear in AI Magazine,
Summer/Fall 1994.

An Introduction to Least Commitment Planning

Daniel S. Weld[1]
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
`weld@cs.washington.edu`

Abstract

Recent developments have clarified the process of generating partially ordered, partially specified sequences of actions whose execution will achive an agent's goal. This paper summarizes a progression of least commitment planners, starting with one that handles the simple STRIPS representation, and ending with one that manages actions with disjunctive precondition, conditional effects and universal quantification over dynamic universes. Along the way we explain how Chapman's formulation of the Modal Truth Criterion is misleading and why his NP-completeness result for reasoning about plans with conditional effects does not apply to our planner.

[1]I thank Franz Amador, Tony Barrett, Darren Cronquist, Denise Draper, Ernie Davis, Oren Etzioni, Nort Fowler, Rao Kambhampati, Craig Knoblock, Nick Kushmerick, Neal Lesh, Karen Lochbaum, Drew McDermott, Ramesh Patil, Kari Pulli, Ying Sun, Austin Tate and Mike Williamson for helpful comments, but retain sole responsibility for errors. This research was funded in part by Office of Naval Research Grant 90-J-1904 and by National Science Foundation Grant IRI-8957302

Contents

1 Introduction

To achieve their goals, agents often need to act in the world. Thus it should be no surprise that the quest of building intelligent agents has forced Artificial Intelligence researchers to investigate algorithms for generating appropriate actions in a timely fashion. Of course the problem is not yet "solved," but considerable progress has been made. In particular, AI researchers have developed two complementary approaches to the problem of generating these actions: *planning* and *situated action.* These two techniques have different strengths and weaknesses, as we illustrate below. Planning is appropriate when a number of actions must be executed in a coherent pattern to achieve a goal or when the actions interact in complex ways. Situated action is appropriate when the best action can be easily computed from the current state of the world (*i.e.*, when no lookahead is necessary because actions do not interfere with each other).

For example, if one's goal is to attend the IJCAI-93 conference in Chambery, France, advanced planning is suggested. The goal of attending IJCAI engenders many subgoals: booking plane tickets, getting to the airport, changing dollars to francs, making hotel reservations, finding the hotel, *etc.* Achieving these goals requires executing a complex set of actions in the correct order, and the prudent agent should spend time reasoning about these actions (and their proper order) in advance. The slightest miscalculation (*e.g.*, attempting to make hotel reservations *after* executing the trans-Atlantic fly action) could lead to failure (*i.e.*, a miserable night on the streets of Paris among the city's many hungry canines).

On the other hand, if the goal is to stay alive while playing a fast paced videogame, advanced planning may be less important. Instead, it may suffice to watch the dangers as they approach and shoot the most threatening attackers first. Indeed, wasting time deliberating about the best target might decrease one's success, since the time would be better spent shooting at the myriad enemy. Domain specific situated-action systems are often implemented as production systems or with hardwired logic (combinational networks). Techniques for automatically compiling these *reactive systems* from declarative domain specifications and learning algorithms for automatically improving their performance are hot topics of research.

In this paper, we neglect the situated techniques and concentrate on the converse approach to synthesizing actions: planning. Planners are characterized by two dimensions which distinguish the construction strategy and the component size respectively (Figure 1). One way of constructing plans is *refinement*, the process of gradually adding actions and constraints; *retraction* eliminates previously added components from a plan, and *transformational planners* interleave refinement and retraction activities. A different dimension concerns the basic blocks that a planner uses when synthesizing a plan: *generative planners* construct plans from scratch while *case-based planners* use a library of previously synthesized plans or plan fragments.[2] Case-based systems are motivated by the observation that many of an agent's actions are routine — for example, when making the daily commute to work or school, one probably executes roughly the same actions in roughly the same order. Even though these actions may interact, one probably doesn't need to think much about the interactions because one has executed similar actions so many times before. The main challenge faced by proponents of a case-based system is developing similarity metrics which allow efficient retrieval of appropriate (previously executed) plans from memory. After all, if you are faced with the task of getting to work but you can't stop thinking about how you cooked dinner last night, then you'll likely arrive rather late.

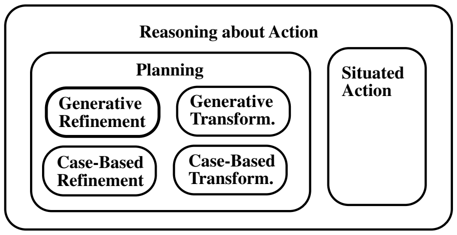


Figure 1: Major approaches for reasoning about action. This paper focuses on generative, refinement planning.

In the next sections, we'll define the planning problem more precisely and then start describing algorithms for solving the problem. We restrict our attention to generative, refinement planning, but our algorithms can be adapted to transformational and case-based approaches [Hanks and Weld, 1992]. As we shall see, planning is naturally formulated as a search problem, but the choice of search space is critical to performance.

2 The Planning Problem

Formally a planning algorithm has three inputs:

1. a description of the world in some formal language,

2. a description of the agent's goal (*i.e.*, what behavior is desired) in some formal language, and

[2]For example, CHEF [Hammond, 1990] and SPA [Hanks and Weld, 1992] are good examples of a transformational case-based planners while PRODIGY/ANALOGY [Veloso and Carbonell, 1993] and PRIAR [Kambhampati and Hendler, 1992] are case-based, refinement planners. All the algorithms presented in the remainder of this paper are generative, refinement algorithms. However, GORDIUS [Simmons, 1988b] provides a good example of a generative, transformational planner (although it can be used in case-based mode as well).

3. a description of the possible actions that can be performed (again, in some formal language). This last description is often called a *domain theory.*

The planner's output is a sequence of actions which, when executed in any world satisfying the initial state description, will achieve the goal. Note that this formulation of the planning problem is *very* abstract. In fact, it really specifies a *class* of planning problems parameterized by the languages used to represent the world, goals, and actions.

For example, one might use propositional logic to describe the effects of actions, but this would preclude describing actions with universally quantified effects. The action of executing a UNIX "`rm *`" command is most naturally described with quantification: "All files in the current directory are deleted." Thus one might describe the effects of actions with first order predicate calculus, but this assumes that all effects are deterministic. It would be very difficult to represent the precise effects of an action such as flipping a coin or prescribing a particular medication for a sick patient (who might or might not get better) without some form of probabilistic representation.

In general, there is a spectrum of more and more expressive languages for representing the world, an agent's goals, and possible actions. The task of writing a planning algorithm is harder for more expressive representation languages, and the speed of the resulting algorithm decreases as well. In this paper, we'll explain how to build planners for several languages, but they all make some simplifying assumptions that we now state.

- **Atomic Time:** Execution of an action is indivisible and uninterruptible, thus we need not consider the state of the world while execution is proceeding. Instead, we may model execution as an atomic transformation from one world state to another. Simultaneously executed actions are impossible.

- **Deterministic Effects:** The effect of executing any action is a deterministic function of the action and the state of the world when the action is executed.

- **Omniscience:** The agent has complete knowledge of the initial state of the world and of the nature of its own actions.

- **Sole Cause of Change:** The only way the world changes is by the agent's own actions. There are no other agents and the world is static by default. Note that this assumption means that the first input to the planner (the world description) need only specify the initial state of the world.

Admittedly, these assumptions are unrealistic. But they *do* simplify the problem to the point where we can describe some simple algorithms. Alternatively, skip ahead to the section on advanced topics where we describe extensions to the algorithms that relax these assumptions.

We'll start our discussion of planning with a very simple language: the propositional STRIPS representation.[3]

The propositional STRIPS representation describes the initial state of world with a complete set of ground literals. For example, the simple world consisting of a table and three blocks shown on the left side of Figure 2 can be described with the following true literals:

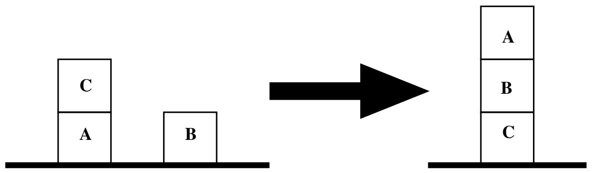


Figure 2: Initial and goal states for the "Sussman anomaly" problem in the blocks world.

```
(on A Table) (on C A) (on B Table) (clear B) (clear C)
```

Since we require the initial state description to be complete, all atomic formulae not explicitly listed in the description are assumed to be false (this is called the "Closed World Assumption" [Reiter, 1980]). This means that `(not (on A C))` and `(not (clear A))` are implicitly in the initial state description as are a bunch of other negative literals.

The STRIPS representation is restricted to *goals of attainment.* In general, a planner might accept an arbitrary description of the behavior desired of the agent over time. For example, one might specify that a robot should cook breakfast but never leave the house. Most planning research, however, has considered goal descriptions that specify features that should hold in the world at the distinguished time point after the plan is executed, even though this renders the "remain in the house" goal inexpressible. Furthermore, the STRIPS representation restricts the type of *goal states* that may be specified to those matching a conjunction of positive literals. For example, the goal situation shown on the right hand side of Figure 2 could be described as the conjunction of two literals `(on B C)` and `(on A B)`. This yields a simple block-stacking challenge called the "Sussman Anomaly."[4]

[3]The acronym "STRIPS" stands for "STanford Research Institute Problem Solver" a very famous and influential planner built in the 1970s to control an unstable mobile robot known affectionately as "Shakey" [Fikes and Nilsson, 1971].

[4]The etymology of the name is a bit puzzling since the problem was discovered at MIT in 1973 by Allen Brown who noticed that the HACKER problem solver had problems dealing

A domain theory, denoted with the Greek letter Λ, forms the third part of a planning problem: it's a formal description of the actions that are available to the agent. In the STRIPS representation, actions are represented with *preconditions* and *effects*. The precondition of each action follows the same restriction as the problem's goal: they are a conjunction of positive literals. An action's effect, on the other hand, is a conjunction that may include both positive and negative literals. For example, we might define the action `move-C-from-A-to-Table` as follows:

- Precondition: `(and (on C A) (clear C))`
- Effect: `(and (on C Table) (not (on C A)) (clear A))`

Actions may be executed only when their precondition is true; in this case we have specified that the robot may move `C` from `A` to the `Table` only when `C` is on top of `A` and has nothing atop it. When an action is executed, it changes the world description in the following way. All the positive literals in the effect conjunction (called the action's *add-list*) are added into the state description while all the negative literals (called the action's *delete-list*) are removed.[5] For example, executing `move-C-from-A-to-Table` in the initial state described above leads to a state in which

```
(on A Table) (on B Table) (on C Table) (clear A) (clear B) (clear C)
```

are true and all other atomic formulae are false.

So that concludes the description of a planner's inputs: a description of the initial state, a description of the goal, and the domain theory, Λ, which is a set of action descriptions. When called with these inputs, a planner should return a sequence of actions that will achieve the goal when executed in the initial state. For example, when given the problem defined by the Sussman

with it. Since HACKER was the core of Gerald Sussman's Ph.D. thesis, he got stuck with the name. In subsequent years, numerous researchers searched for elegant ways to handle it. Tate's INTERPLAN system [Tate, 1975] used more sophisticated reasoning about goal interactions to find an optimal solution and Sacerdoti's NOAH planner [Sacerdoti, 1975] introdcued a more flexible representation to sidestep the problem. Because the planners described in this paper adopt these techniques, they have no problem with the "Anomalous Situation." Still it's worth explaining why the problem flummoxed early researchers. Note that the problem has two "subgoals:" to achieve `(on A B)` and to achieve `(on B C)`. It seems natural to try divide and conquer, but if we try to achieve the first subgoal before starting the second then the obvious solution is to put `C` on the `Table` then put `A` on `B` and we accidentally wind up with `A` on `B` when `B` is still on the `Table`. Of course one can't get `B` on `C` without taking `A` off `B` so trying to solve the first subgoal first appears to be a mistake. But if we try to achieve `(on B C)` first, then we have a similar problem: `B`'s on `C` while `A` is still buried at the bottom of the stack. So no matter which order is tried, the subgoals interfere with each other. But humans seem to use divide and conquer, so why can't computers? In fact, they can as we show in the section on plan-space search.

[5]It's illegal for an action's effect to include an atomic fomula and its negation, since this would lead to an undefined result.

anomaly's initial and goal states (Figure 2) and a set of actions like the one described above, we'd like our planner to return a sequence like:

```
move-C-from-A-to-Table
move-B-from-Table-to-C
move-A-from-Table-to-B
```

As we shall see, there are a variety of algorithms that do exactly this, but some are much more efficient than others. We'll start by looking at planners which are conceptually quite simple, and then look at more sophisticated ways of planning.

3 Search through World Space

The simplest way to build a planner is to cast the planning problem as search through the space of world states (shown in Figure 3). Each node in the graph denotes a state of the world, and arcs connect worlds that can be reached by executing a single action. In general, arcs are directed, but in our encoding of the Blocks World, all actions are reversible so we have replaced two directed edges with a single arc to increase readability. Note that the initial and goal world states of the Sussman anomaly are highlighted in grey. When phrased in this manner, the solution to a planning problem (*i.e.*, the *plan*) is a path through state-space. Note that the three-step solution listed at the end of the previous section is the shortest path between these two states, but many other paths are possible.

The advantage of casting planning as a simple search problem is the immediate applicability of all the familiar brute force and heuristic search algorithms [Korf, 1988]. For example, one could use depth-first, breadth-first, or iterative deepening A* search starting from the initial state until the goal is located. Alternatively, more sophisticated, memory bounded algorithms could be used [Russell, 1992, Korf, 1992]. Since the tradeoffs between these different searching algorithms have been discussed extensively elsewhere, we focus instead on the *structure* of the search space. A handy way to do this is to specify the planner with a *nondeterministic algorithm.* This idea may seem strange at first, but we'll use it extensively in subsequent sections, so it's important to learn it now. In fact, it's quite simple: when specifying the planning algorithm one uses a nondeterministic **choose** primitive. **choose** takes a set of possible options and magically selects the right one. The beauty of this nondeterministic primitive lies in its ease of implementation: **choose** can be simulated with any conservative search method or it can be approximated with an aggressive search strategy. By decoupling the search strategy from the basic nondeterministic algorithm two things are accomplished: (1) the algorithm becomes simpler and easier to understand, and (2) the imple-

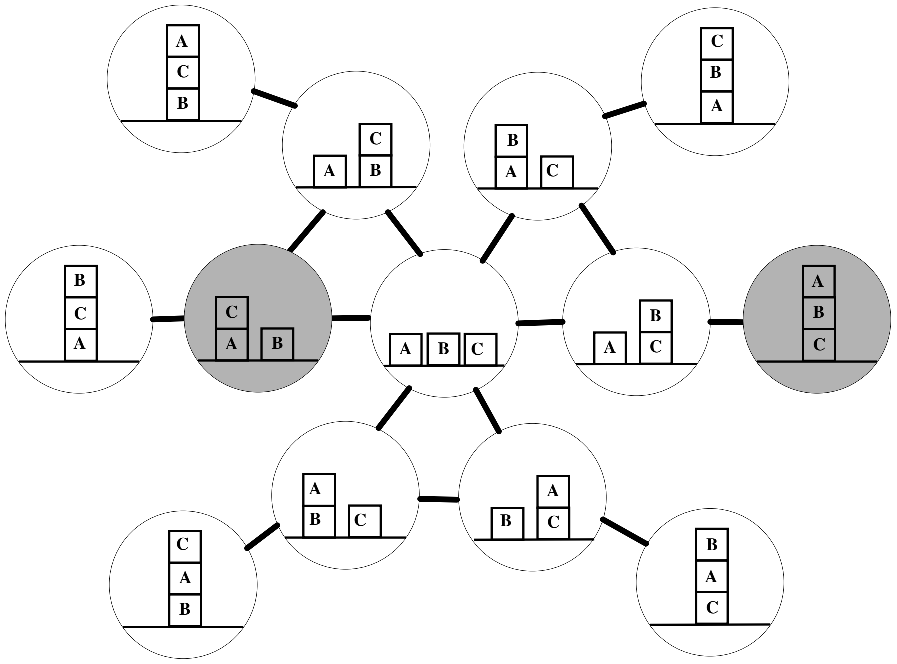


Figure 3: World Space.

mentor can easily switch between different search strategies in an effort to improve performance.

3.1 Progression

To make this concrete, Figure 4 contains a simple nondeterministic planner that operates by searching forward from the initial world-state until it finds a state in which the goal specification is satisfied.

The right way to think about a nondeterministic algorithm is with *you* personally calling the shots every time that **choose** gets called. For example, if we try ProgWS on the Sussman anomaly, then the first call to the procedure has world-state set to the initial state (the leftmost grey state in Figure 3), goal-list set to the implicit conjunction `((on A B) (on B C))`, and path set to the null sequence. Since the initial state doesn't satisfy the goal, execution falls to line 3 and **choose** is called. A moment's thought should convince you that the best choice makes `Move-C-from-A-to-Table` be the first action, so let's assume that this is the choice made by the computer. By giving the program a magical oracle it can easily find the sequence of three correct choices that will lead to a solution. Since we assume that the oracle always makes the best choice, the program can quit (confident that no solution is possible) if it ever runs into a dead end.

Of course, if one wants to implement ProgWS on any of the (nonmagical) computers that exist today (and if one doesn't want to get a lot of email from the program asking for advice!) then one needs to use search. A simple

```
Algorithm: ProgWS(world-state, goal-list, $\Lambda$, path)
  1. If world-state satisfies each conjunct in goal-list,
  2. Then return path,
  3. Else let Act = choose from $\Lambda$ an action whose precondition is satisfied
     by world-state:
     (a) If no such choice was possible,
     (b) Then return failure,
     (c) Else let S = the result of simulating execution of Act in world-state
         and return ProgWS(S, goal-list, $\Lambda$, Concatenate(path, Act)).
```

Figure 4: ProgWS: a Progressive, World-State planner. The initial call should set path to the null sequence.

technique is to implement **choose** with breadth-first search. This way, even though it wouldn't have the oracle, the planner could try all paths in parallel (storing them on a queue and time slicing between them) until it found a state that satisfied the goal specification. Any time a nondeterministic algorithm would find a solution, the breadth-first search version will also (although in the worst case it might take the searching version exponentially longer).

3.2 Regression

Figure 4 describes just one way to convert planning to a search through the space of world states. Another approach, called regression planning [Waldinger, 1977], is outlined in Figure 5. Instead of searching forward from the initial state (which is what ProgWS does), the RegWS algorithm (adapted from [Nilsson, 1980]) searches backwards from the goal. Intuitively, RegWS reasons as follows: "I want to eat, so I need to cook dinner, so I need to have food, so I need to buy food, so I need to go to the store..." At each step it **choose**s an action that might possibly help satisfy one of the outstanding goal conjuncts.

To illustrate RegWS on a more concrete example, the Sussman anomaly, then cur-goals is initially set to the list of conjuncts `((on A B) (on B C))`. The first call to **choose** demands an action whose effect contains a conjunct that appears in cur-goals. Since the action `Move-A-from-Table-to-B` has the effect of achieving `(on A B)`, let's assume that the planner magically (nondeterministically) makes that choice.

The next step, called *goal regression*, forms the core of the `RegWS` algorithm: G is assigned the result of regressing a logical sentence (the conjunction corresponding to the list cur-goals) through the action Act. The result

```
Algorithm: RegWS(init-state, cur-goals, $\Lambda$, path)
  1. If init-state satisifes each conjunct in cur-goals,
  2. Then return path,
  3. Else do:
     (a) Let Act = choose from $\Lambda$ an action whose effect matches at least
         one conjunct in cur-goals.
     (b) Let G = the result of regressing cur-goals through Act.
     (c) If no choice for Act was possible or G is undefined, or G $\supset$ cur-goals,
     (d) Then return failure,
     (e) Else return RegWS(init-state, G, $\Lambda$, Concatenate(Act, path)).
```

Figure 5: RegWS: a Regressive, World-State planner. The initial call should set path to the null sequence.

of this regression is another logical sentence that encodes the weakest preconditions that must be true *before* Act is executed in order to assure that cur-goals will be true *after* Act is executed. This is simply the union of Act's preconditions with all the current goals except those provided by the effects of Act:

$$\text{preconditions(Act)} \cup (\text{cur-goals} - \text{goals-added-by(Act)})$$

In our example, Act has `(on A Table)` as its precondition and `(and (on A B) (not (on A Table)))` as its effect so the result of regressing `(and (on A B) (on B C))` is: `(and (on A Table) (on B C))`.

Since Act achieved `(on A B)`, regression removed that literal from the sentence (replacing it with the precondition of Act, namely `(on A Table)`. Since Act doesn't affect the other goal, `(on B C)`, it remains part of the weakest precondition. Note that the sentence produced by regression is still a conjunction; this is guarenteed to be true as long as action preconditions are restricted to conjunctions, so it is ok to encode G and cur-goals with lists.

The next line of RegWS is also interesting — it says that if **choose** can't find an action whose regression satisfies certain criteria, then a dead end has been reached. There are three parts to the dead-end check, and we discuss them in turn:

1. If no action has an effect containing a conjunct that matches one of the conjuncts in cur-goals, then no action is profitable. To see why this is the case, note that unless Act has a matching conjunct, the result of performing goal regression will be a strictly *larger* conjunctive sentence! Whenever G is satisfied by the initial state, then cur-goals will be too.

Thus there is no point in considering such an Act because any successful plan that might result could be improved by eliminating it from the path.

2. If the result of regressing cur-goals through Act is to make G undefined then any plan that adds Act to this point in the path will fail. What might make G undefined? Recall that regression returns the weakest preconditions that must be true *before* Act is executed in order to make cur-goals true *after* execution. But what if one of Act's effects directly conflicts with cur-goals? That would make the weakest precondition undefined because *no matter* what was true before Act, execution would ruin things. A good example of this results when one tries to regress `((on A B) (on B C))` through `Move-A-from-B-to-Table`. Since this action negates `(on A B)`, the weakest preconditions are undefined.

3. If G $\supseteq$ cur-goals then there's really no point in adding Act to path for the same reasons that were explained in bullet 1 above. In fact, one can show that G $\supset$ cur-goals whenever the action's effect doesn't match any conjunct in cur-goals, but the converse is false. Thus, strictly speaking, the G $\supset$ cur-goals renders the test of bullet 1 unnecessary; however, eliminating it would result in reduced efficiency since many more regressions would be required.

3.3 Analysis

Our presentation of the ProgWS and RegWS planning algorithms suggests several natural questions. The first questions concern the soundness (*i.e.*, if a plan is returned, will it really work?) and completeness (*i.e.*, if a plan exists, does a sequence of nondeterministic choices exist that will find it?) of the algorithms. Although we won't prove it here, both algorithms are sound and complete.

The most important question, however, is "Which algorithm is faster?" In their nondeterministic forms, of course, they have the same complexity: with perfect luck, they'll each make the same number (say n) of nondeterministic choices before finding a solution. However, a real implementation must use search to implement the nondeterminism, so an important question is "How many choices must be considered at each nondeterministic branch point?" Let's call this number b. Even a small difference in b can lead to a tremendous difference in planning efficiency since brute-force searching time is $O(b^n)$.

If one grants the plausible assumption that the *goal* of a planning problem is likely to involve only a small fraction of the literals used to describe the state, then regession planning is likely to have a much smaller branching factor at each call to **choose**; as a result it's likely to run much faster. To see this, note that there will probably be many actions that could be executed in the initial state but only a few that are relevant to the goal (*i.e.*, have effects that match the goal and have legal regressions). Since PROGWS must consider *all actions whose preconditions are satisfied by the initial state*, it can't benefit from the guidance provided by the planning objective.

In some cases, of course, the situation may be reversed. And we should note that there are a variety of other search techniques (means-ends analysis, bidirectional search, etc.) that we haven't discussed.[6] The reason for this selective portrayal stems from the nature of world space search itself. As the next section shows, it's often much better to search the space of partially specified plans.

4 Search through the Space of Plans

In 1974, Earl Sacerdoti built a planner, called NOAH, with many novel features. The innovation we'll focus on here is the reformulation of planning from one search problem to another. Instead of searching the space of world states (in which arcs denote action execution), Sacerdoti phrased planning as search through *plan space*.[7] In this space, nodes represent partially-specified plans and edges denote *plan refinement operations* such as the addition of an action to a plan. Figure 6 illustrates one such space. Once again, the initial and goal state are highlighted in grey. The initial state represents the *null plan* which has no actions, and the goal state represents a complete, working plan for the Sussman anomaly. Note that while world state planners had to return the *path* between initial and goal states, in plan-space the goal state *is* the solution.

4.1 Total Order Planning

At this point we are forced to confess. We've claimed that it's useful to think of planning as search through plan-space and we've explained that in plan-space nodes denote plans, but we haven't said what plans really *are*. In fact this is a very subtle issue that we shall discuss in some depth, but for now let's consider a simple answer and suppose that a plan is represented as a totally-ordered sequence of actions. In that case, we can view the familiar REGWS algorithm as *isomorphic* to a plan-space planner! After all, at every recursive call, it passes along an argument, `path`, which is a totally ordered

[6]Means-ends analysis, the problem solving strategy used by GPS [Newell and Simon, 1963] is especially important, both from a historical perspective and because of its ubiquity in machine learning research on speedup learning [Minton, 1988, Minton *et al.*, 1989b]. Unfortunately, GPS-like planners are incomplete (for example they cannot solve the Sussman anomaly) which complicates analysis and comparison to the algorithms in this paper. Future work is needed to investigate the benefits, if any, of the GPS approach.

[7]In fact, NOAH didn't actually *search* the space in any exhaustive manner (*i.e.* unlike NONLIN [Tate, 1977] it did no backtracking), but it is still credited with reformulating the space in question.

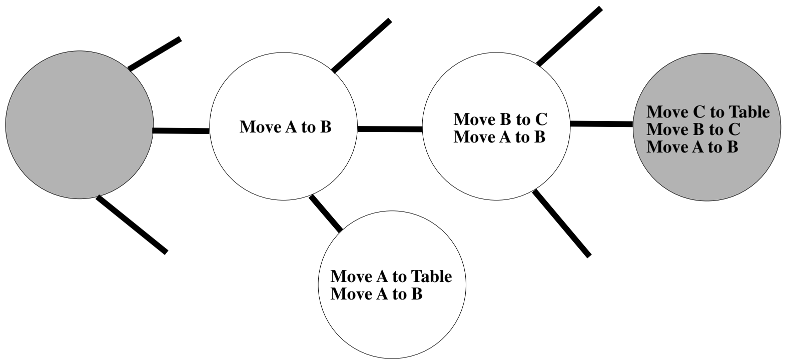


Figure 6: Plan Space

sequence of actions (*i.e.*, a plan). In fact, if we watch the successive values of `path` at each recursive call, we get the picture shown in Figure 6.

In summary, the nature of the space being searched by an algorithm is (somewhat) in the eye of the beholder. If we view REGWS as searching the space of world-states, it's a regression planner. If we view it as searching the space of totally ordered plans, then the plan refinement operators modify the current plan by prepending new actions to the beginning of the sequence.

So what's the point of thinking about planning as a search process through plan-space? This framework facilitates thinking about alternative plan refinement operators and leads to more powerful planning algorithms. For example, by adding new actions into the plan *at arbitrary locations* one can devise a planner that works much better than one which is restricted to prepending the actions. However, we won't describe that algorithm here because it's possible to do even better by changing the plan representation itself, as described in the next section.

4.2 Partial Order Planning

Think for a moment about how *you* might solve a planning problem. For concreteness, we return to the introductory example of planning a trans-Atlantic trip to IJCAI-93. To make the trip, one needs to purchase plane tickets and also to buy a "Let's Go" guide to France (to enable choosing hotels and itinerary). However, there's no need to decide (*yet*) which purchase should be executed first. This is the idea behind least commitment planning — to represent plans in a flexible way that enables deferring decisions. Instead of committing prematurely to a complete totally ordered sequence of actions, plans are represented as a partially ordered sequence and the planning algorithm practices "Least Commitment" — only the essential ordering decisions are recorded.

4.2.1 Plans, Causal Links & Threats

We represent a plan as a three-tuple: $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$ in which $\mathcal{A}$ is a set of **a**ctions, $\mathcal{O}$ is a set of **o**rdering constraints over $\mathcal{A}$, and $\mathcal{L}$ is a set of causal links (described below). For example, if $\mathcal{A} = \{A_1, A_2, A_3\}$ then $\mathcal{O}$ might be the set $\{A_1 < A_3, A_2 < A_3\}$. These constraints specify a plan in which A_3 is necessarily the last (of three) actions, but does not commit to a choice of which of the three actions comes first. Note that these ordering constraints are *consistent* because there exists at least one total order that satisfies them. As least commitment planners refine their plans, they must do constraint satisfaction to ensure the consistency of $\mathcal{O}$. Maintaining the consistency of a partially ordered set of actions is just one (simple) example of constraint satisfaction in planning — we'll see more in subsequent sections.

A key aspect of least commitment is keeping track of past decisions and the reasons for those decisions. For example, if you purchase plane tickets (to satisfy the goal of boarding the plane) then you should be sure to take them to the airport. If another goal (having your hands free to open the taxi door, say) causes you to drop the tickets, you should be sure to pick them up again. A good way of ensuring that the different actions introduced for different goals won't interfere is to record the dependencies between actions explicitly.[8] To record these dependencies, we use a data structure, called a *causal link*, that was invented by Austin Tate for use in the NONLIN planner [Tate, 1977]. A causal link is a structure with three fields: two contain pointers to plan actions (the link's producer, A_p, and its consumer, A_c); the third field is a proposition, Q, which is both an effect of A_p and a precondition of A_c. We write such a causal link as $A_p \xrightarrow{Q} A_c$ and store a plan's links in the set $\mathcal{L}$.

Causal links are used to detect when a newly introduced action interferes with past decisons. We call such an action a *threat.* More precisely, suppose that $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$ is a plan and $A_p \xrightarrow{Q} A_c$ is a causal link in $\mathcal{L}$. Let A_t be a different action in $\mathcal{A}$; we say that A_t *threatens* $A_p \xrightarrow{Q} A_c$ when the following two criteria are satisfied:

- $\mathcal{O} \cup \{A_p < A_t < A_c\}$ is consistent, and
- A_t has $\neg Q$ as an effect.

For example, if A_p asserts $Q =$`(on A B)`, which is a precondition of A_c, and the plan contains $A_p \xrightarrow{Q} A_c$, then A_t would be considered a threat if it moved `A` off `B` and the ordering constraints didn't prevent A_t from being executed between A_p and A_c.

When a plan contains a threat, then there is a danger that the plan won't work as anticipated. To prevent this from happening, the planning algorithm

[8]An alternative approach is to repeatedly compute these interactions, but this is often less efficient.

must check for threats and take evasive countermeasures. For example, the algorithm could add an additional ordering constraint to ensure that A_t is executed before A_p. This particular threat protection method is called *demotion*; adding the symmetric constraint $A_c < A_t$ is called *promotion*.[9] As we'll see in subsequent sections, there are other ways to protect against threats as well.

4.2.2 Representing Planning Problems as Null Plans

Uniformity is the key to simplicity. It turns out that the simplest way to describe a plan-space planning algorithm is to make it use one uniform representation for both planning problems and for incomplete plans. The secret to achieving this uniformity is an encoding trick: the initial state description and goal conjunct can be bundled together into a special three-tuple called the *null plan*.

The encoding is very simple. The null plan of a planning problem has two actions, $\mathcal{A} = \{A_0, A_\infty\}$, one ordering constraint, $\mathcal{O} = \{A_0 < A_\infty\}$, and no causal links: $\mathcal{L} = \{\}$. All the planning activity stems from these two actions. A_0 is the "*start*" action — it has no preconditions and its effect specifies which propositions are true in the planning problem's initial state and which are false.[10] A_∞ is the "*end*" action — it has no effects, but its precondition is set to be the conjunction from the goal of the planning problem. For example, the null plan corresponding to the Sussman anomaly is shown in Figure 7.

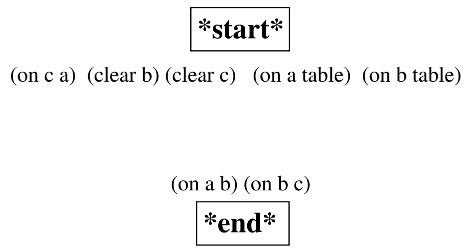


Figure 7: The null plan for the Sussman Anomaly contains two actions — the "*start*" action precedes the "*end*" action.

[9]The rationale behind the names stems from the fact that demotion moves the threat *lower* in the temporal ordering while promotion moves it higher.

[10]Actually, we adopt the convention that every proposition which is not explicitly specified to be true in the initial state is assumed to be false. This is called the *closed world assumption* or CWA.

4.2.3 The POP Algorithm

We now describe a simple, regressive algorithm that searches the space of plans.[11] POP starts with the null plan for a planning problem and makes nondeterministic choices until all conjuncts of every action's precondition have been supported by causal links and all threatened links have been protected from possible interference. The ordering constraints, $\mathcal{O}$, of this final plan may still specify only a partial order — in this case, any total order consistent with $\mathcal{O}$ is guaranteed to be an action sequence that solves the planning problem. See Figure 8 — the first argument to POP is a plan structure and the second is an agenda of goals that need to be supported by links. Each item on the agenda is represented as a pair $\langle Q, A_i \rangle$ where Q is a conjunct of the precondition of A_i. (Note: many times the identity of A_i will be clear from the context and we will pretend that **agenda** contains propositions, such as Q, instead of $\langle Q, A_i \rangle$ pairs.)

It's very important to understand how this algorithm works in detail, so we now illustrate its behavior on the Sussman anomaly. When making the initial call to POP, we provide two arguments: the null plan, shown in Figure 7, and **agenda** $= \{\langle$`(on A B)`$, A_\infty \rangle, \langle$`(on B C)`$, A_\infty \rangle\}$. Since **agenda** isn't empty, control passes to line 2. There are two choices for the immediate goal: either $Q =$`(on A B)` or $Q =$`(on B C)` so POP must make a choice. Now comes a crucial, but subtle point. POP has to choose between the two subgoals, but this was not written with **choose**— why not? The answer is that the choice does not matter as far as *completeness* is concerned — eventually both choices must be made. As a result there is no reason for a searching version of the program to backtrack over this choice. Does this mean that the choice doesn't matter? Absolutely not! One choice might lead the planner to find an answer very quickly while the other choice would lead to enourmous search. In practice, the choice can be very important for efficiency and it is often useful to interleave reasoning about different subgoals. But the order in which subgoals are considered by the planner does *not* affect completeness and it is *not* important to a nondeterministic algorithm — the same number of nondeterministic choices will be made either way.

Anyway, suppose POP selects `(on B C)` from the **agenda** as the goal to work on first; A_{need} is set to A_∞. Line 3 needs to **choose** (a real nondeter-

[11] The POP planner is very similar to McAllester's SNLP algorithm [McAllester and Rosenblitt, 1991] which is an improved formalization of Chapman's TWEAK planner [Chapman, 1987]. The difference between SNLP and POP concerns the definition of threat. SNLP treats A_t as a threat to a link $A_p \xrightarrow{Q} A_c$ when A_t has Q as an effect as well as when it has $\neg Q$ as an effect. Although this may seem counterintuitive (what does it matter if Q is asserted twice?), this definition leads to a property, called *systematicity*, which reduces the overall size of the search space. It's widely believed that systematicity is interesting from a technical point of view, but does not necessarily lead to increased planning speed. See [Kambhampati, 1993a] for a discussion.

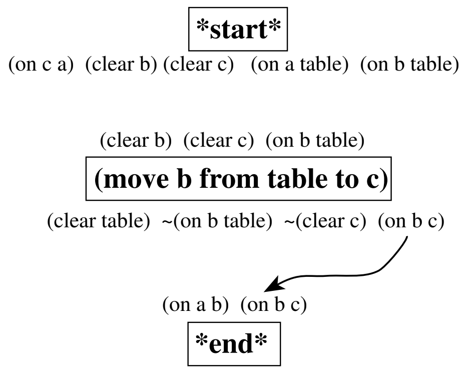
```
Algorithm: POP($\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$, agenda, $\Lambda$)

  1. Termination: If agenda is empty, return $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$.
  2. Goal selection: Let $\langle Q, A_{need} \rangle$ be a pair on the agenda (by definition
     $A_{need} \in \mathcal{A}$ and $Q$ is a conjunct of the precondition of $A_{need}$).
  3. Action selection: Let $A_{add}$ = choose an action that adds $Q$ (either
     a newly instantiated action from $\Lambda$, or an action already in $\mathcal{A}$ which
     can be consistently ordered prior to $A_{need}$). If no such action exists
     then return failure. Let $\mathcal{L}' = \mathcal{L} \cup \{A_{add} \xrightarrow{Q} A_{need}\}$, and let $\mathcal{O}' = \mathcal{O} \cup$
     $\{A_{add} < A_{need}\}$. If $A_{add}$ is newly instantiated, then $\mathcal{A}' = \mathcal{A} \cup \{A_{add}\}$
     and $\mathcal{O}' = \mathcal{O}' \cup \{A_0 < A_{add} < A_\infty\}$ (otherwise let $\mathcal{A}' = \mathcal{A}$).
  4. Update goal set: Let agenda' = agenda $-$ $\{\langle Q, A_{need} \rangle\}$. If $A_{add}$ is
     newly instantiated, then for each conjunct, $Q_i$, of its precondition add
     $\langle Q_i, A_{add} \rangle$ to agenda'.
  5. Causal link protection: For every action $A_t$ that might threaten a
     causal link $A_p \xrightarrow{R} A_c \in \mathcal{L}'$ choose a consistent ordering constraint, either
       (a) Demotion: Add $A_t < A_p$ to $\mathcal{O}'$, or
       (b) Promotion: Add $A_c < A_t$) to $\mathcal{O}'$.
     If neither constraint is consistent, then return failure.
  6. Recursive invocation: POP($\langle \mathcal{A}', \mathcal{O}', \mathcal{L}' \rangle$, agenda', $\Lambda$).
```

Figure 8: POP: a regressive Partial Order Planner. The initial call must set $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$ to the null plan for the problem and set AGENDA to the list of goal conjuncts.

ministic choice, this time!) an action, A_{add}, which has `(on B C)` as an effect. Suppose that the magic oracle suggests making A_{add} be a new instance of a `move-B-from-Table-to-C` action. A new causal link, $A_{add} \xrightarrow{\texttt{(on B C)}} A_\infty$, is added to $\mathcal{L}'$, and the **agenda** is updated. Since there are no threats to the sole link, line 6 makes a recursive call with the arguments depicted in Figure 9.

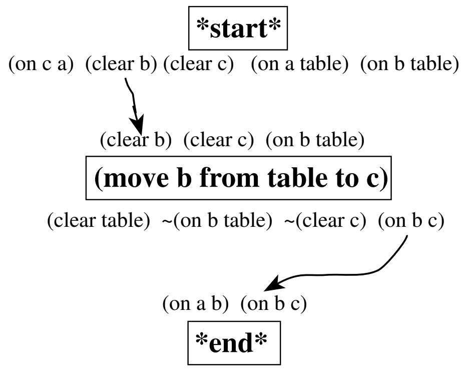
On the second invocation of POP **agenda** is still not empty, so another goal must be chosen. Suppose that the `(clear B)` conjunct of the recently added `move-B-from-Table-to-C` action's precondition is selected as Q in line 2. Next, in line 3 **choose** is called to make the nondeterministic choice of a producing action. Suppose that instead of instantiating a new action (as we illustrated last time), the planner decides to reuse an existing one: the "*start*" action A_0. The net effect of this pass through POP is to add a single link to $\mathcal{L}$ as illustrated in Figure 10 and to shrink the **agenda** slightly.

Suppose that on the third invocation of POP, the planner selects the top-



start
(on c a) (clear b) (clear c) (on a table) (on b table)
(clear b) (clear c) (on b table)
(move b from table to c)
(clear table) ~(on b table) ~(clear c) (on b c)
(on a b) (on b c)
end

Figure 9: After adding a causal link to support `(on B C)`, the plan is as shown and `agenda` contains {`(clear B) (clear C) (on B Table) (on A B)`} as open propositions.



start
(on c a) (clear b) (clear c) (on a table) (on b table)
(clear b) (clear c) (on b table)
(move b from table to c)
(clear table) ~(on b table) ~(clear c) (on b c)
(on a b) (on b c)
end

Figure 10: After adding a causal link to support `(clear B)`, the plan has two causal links and `agenda` is set to {`(clear C) (on B Table) (on A B)`}.

level goal `(on A B)` from the `agenda`. Once again, several possibilities exist for the nondeterministic choice of line 3. Suppose that POP decides to instantiate a new `move-A-from-Table-to-B` action as A_{add}. A new causal link gets added to $\mathcal{L}$, the new action is constrained to precede A_∞, and the `agenda` is updated. Things get a bit more interesting when control flow reaches line 5. Note that both of the new actions, `move-A-from-Table-to-B` and `move-B-from-Table-to-C`, are constrained to precede A_∞ but $\mathcal{O}$ contains no constraints on their relative ordering. Furthermore, note that `move-A-from-Table-to-B` negates `(clear B)`. But this means that it threatens the link from A_0 labeled `(clear B)`. This is illustrated in Figure 11.

To protect against this threat, POP must nondeterministically **choose** an ordering constraint. In general, there are two possibilities: either constrain the `move-A` action after the `move-B` action or constrain `move-A` to precede the "*start*" action A_0. However, since line 3 assures that every action follows A_0, this last choice would make $\mathcal{O}$ inconsistent. Thus, POP orders the threat after the link's consumer as shown in Figure 12.

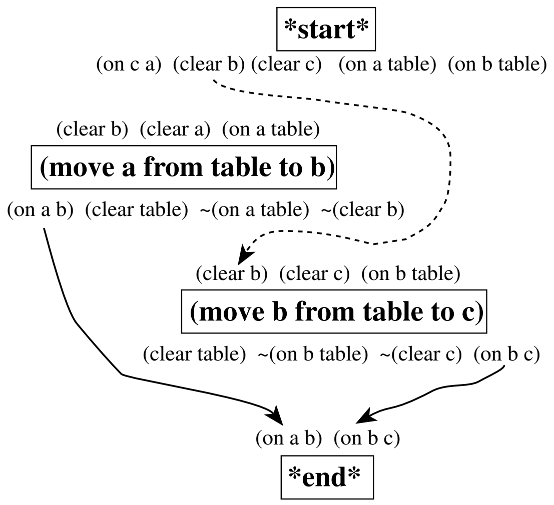

start
(on c a) (clear b) (clear c) (on a table) (on b table)
(clear b) (clear a) (on a table)
(move a from table to b)
(on a b) (clear table) ~(on a table) ~(clear b)
(clear b) (clear c) (on b table)
(move b from table to c)
(clear table) ~(on b table) ~(clear c) (on b c)
(on a b) (on b c)
end

Figure 11: Since the `move-A` action could possibly precede the `move-B` action, it threatens the link labeled `(clear B)` as indicated by the dashed line.

Since the `agenda` still contains five entries, there is much work left to be done. However, all subsequent decisions follow the same lines of reasoning as we have shown above so we omit them here. Eventually, POP returns the plan shown in Figure 13. Careful inspection of this figure confirms that no link is threatened. Indeed, the three actions in $\mathcal{A}$ (besides the dummies A_0 and A_∞ are exactly the same as the ones returned by the world-state planners of the previous section. Like those planners, one can prove that POP is sound and complete.

4.2.4 Implementation Details

To implement POP one must choose data structures to represent the partial order over actions ($\mathcal{O}$). The operations that the data structure needs to support are the addition of new constraints to $\mathcal{O}$, testing if $\mathcal{O}$ is consistent, determining if A_i can be consistently ordered prior to A_j, and returning the set of actions that can be ordered before A_j. In fact, this set of interface operations can be reduced to the ability to add or delete $A_i < A_j$ from $\mathcal{O}$, and test $\mathcal{O}$ for consistency, but this won't necessarily lead to the greatest efficiency since many more queries are typically performed (*i.e.*, in threat detection as discussed in the paragraph below) than there are true updates. Caching the results of queries (*i.e.*, incrementally computing the transitive closure) can significantly increase performance. If a denotes the number of actions in a plan, it takes $O(a^3)$ time to compute the transitive closure and $O(a^2)$ space to store it, but queries can be answered quickly (see the Floyd-Warshall algorithm and discussion [Cormen *et al.*, 1991, p562]). Considerable research has focussed on this time-space tradeoff and on variants which assume a different interface to the temporal manager (see [Williamson and Hanks,

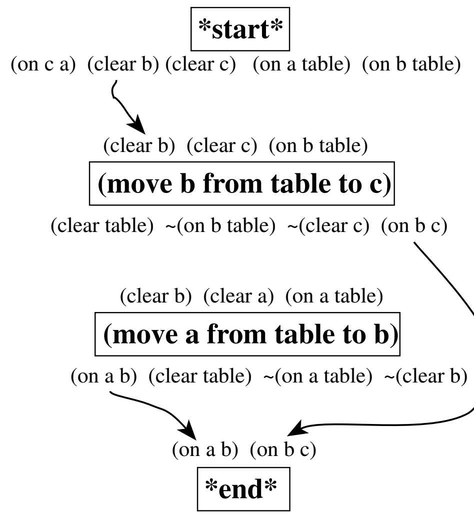

start
(on c a) (clear b) (clear c) (on a table) (on b table)
(clear b) (clear c) (on b table)
(move b from table to c)
(clear table) ~(on b table) ~(clear c) (on b c)
(clear b) (clear a) (on a table)
(move a from table to b)
(on a b) (clear table) ~(on a table) ~(clear b)
(on a b) (on b c)
end

Figure 12: After promoting the threatening action, the plan's actions are totally ordered.

1993] for a nice discussion and pointers).

Another implementation detail concerns the efficency of testing for threatened causal links. There can be $O(a^2)$ links and hence $O(a^3)$ threats. We've found that the most efficient way to handle threats is incrementally:

- Whenever a new causal link is added to $\mathcal{L}$, all actions in $\mathcal{A}$ are tested to see if they threaten it. This takes $O(a)$ time.
- Whenever a new action instance is added to $\mathcal{A}$, all links in $\mathcal{L}$ are tested to see if they are threatened. This takes $O(a^2)$ time.

4.3 Analysis

In general, the expected performance of a search algorithm is $O(cb^n)$. The three parameters that determine performance are explained below:

1. How many times is nondeterministic **choose** called before a solution is obtained? This determines the exponent n.
2. How many possibilities need to be considered (by a searching algorithm) at each call to **choose**? This determines the average branching factor b.
3. How long does it take to process a given node in the search space (*i.e.*, how much processing goes on before the recursive call)? We have written this as the constant c although it is usually a function of the size of the node being considered.

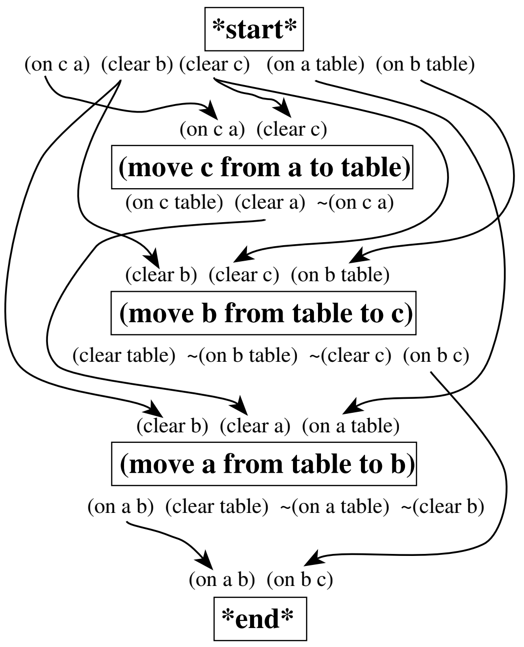

start
(on c a) (clear b) (clear c) (on a table) (on b table)
(on c a) (clear c)
(move c from a to table)
(on c table) (clear a) ~(on c a)
(clear b) (clear c) (on b table)
(move b from table to c)
(clear table) ~(on b table) ~(clear c) (on b c)
(clear b) (clear a) (on a table)
(move a from table to b)
(on a b) (clear table) ~(on a table) ~(clear b)
(on a b) (on b c)
end

Figure 13: Eventually, this plan is returned as a solution

The cost per node in the search space, c, is quite different for REGWS and POP, but this doesn't matter much in practice. For the world state planner, the per-node operations can all be implemented in time proportional to the number of current goals, |**cur-goals**|, and the number and complexity of actions. POP, on the other hand, has several operations (testing for threatened links, for example) whose complexity grows with the length of the plan under consideration. We say that these factors don't matter very much because the exponential b^n dominates these costs.

The number of nondeterministic calls, n, can vary somewhat. In particular, REGWS makes one call per action introduced (*i.e.*, $n = a$) while POP makes one per precondition conjunct (actually more if links are threatened). Ignoring the issue of threatened links, POP will have a higher n if a given action supports more than one precondition conjunct, and this is almost always the case. For example, the "*start*" action often supports many conjuncts. On the other hand, the ratio between POP's n and REGWS's n can never be greater than the maximum number of precondition conjuncts per action, and this is typically a small constant (3 for the blocks world). In any case, the value of n certainly doesn't suggest that POP will run faster.

But POP usually *does* run faster, and b is the reason why. POP achieves completeness with a much smaller branching factor than the world-space algorithms. At each call to **choose** in line 3, POP has to consider only those actions whose effects are relevant to the particular goal, Q, chosen in line 2. Recall that this choice of Q does not require backtracking. The situation with REGWS is quite different. Line 3a's nondeterministic choice must consider all actions whose effects are relevant to *any* member of cur-goals. In other words, REGWS has to backtrack over the choice of which goal (Q) to work on next; failure to consider all possibilities would sacrifice completeness. The reason for this stems from the fact that REGWS links the decision of which *goal* to work on next with the decision of *when to execute* the resulting actions. By using a least commitment approach with the set of ordering constraints, $\mathcal{O}$, POP achieves a branching factor that is smaller by a factor equal to the average size of cur-goals (or agenda), which can grow quite big. Indeed, the increased branching factor is usually the dominant effect.

There are many other factors involved, and a detailed analysis is complex. See [Barrett and Weld, 1993, Barrett and Weld, 1994, Minton *et al.*, 1991, Minton *et al.*, 1992] for different types of analytic comparisons and experimental treatments.

It is also important to note that the POP algorithm represents just one point on the spectrum of possible least commitment planning algorithms. Brevity precludes a discussion of other interesting possibilites, but see [Kambhampati, 1993a, Kambhampati, 1993b] for a survey of approaches and a fascinating taxonomy of design tradeoffs.

5 Action Schemata with Variables

Since the idea of least commitment has proven useful, it is natural to wonder if one can take it further. Indeed, this is both possible and useful! Before we describe the next step, we wish to highlight the relationship between least commitment and constraint satisfaction. Note that the key step in allowing POP to delay decisions about when individual actions are to be scheduled was the inclusion of the set of ordering constraints, $\mathcal{O}$, and the attendant constraint satisfaction algorithms for determining consistency. It turns out that we can perform the same trick when choosing which action to use to support an open condition: delay the decision by adding constraints and gradually refining them.

Take a look back at Figure 9 in which POP has just created its first causal link from a new `move-B-from-Table-to-C` action to support the goal `(on B C)`. How did this choice get made? Line 3 of POP selected between all existing (there were none) and new actions that had `(on B C)` as an effect. What were the other possibilities? Well, a `move-B-from-A-to-C` action would have worked too. And if there were other blocks mentioned in the problem, then POP would have had to consider moving `B` from `D` or from `E` or .... But this is absurd! Why (at this point in the planning process) should POP have to worry about where `B` is going to be? Instead, it is much better to delay that commitment until later (when some choices may be easily ruled out).

We can accomplish this by having POP add the action `Move-B-from-?x-to-C` where `?x` denotes a variable whose value has not yet been chosen. In fact, why not go all the way and define a general `move` schema which defines the class of actions that move an arbitrary block, `?b`, from an arbitrary prior location, `?x`, to any destination `?y`? We'll call such an action schema an *operator*. When choosing to instantiate this operator, one could specify that `?b`=`B` and `?y`=`C`. Subsequent decisions could add more and more constraints on the value of `?x` until eventually it has a unique value. The key question is "What types of constraints should be allowed?" The simplest[12] answer is to allow *codesignation* and *noncodesignation* constraints, which we write as `?x=?y` and `?x≠?y`, respectively. To make these ideas concrete, see the definition of Figure 14 which defines a `move` operator which is *reasonably* general.[13]

```
(define (operator move)
    :parameters   (?b ?x ?y)
    :precondition (and (on ?b ?x) (clear ?b) (clear ?y)
                       (≠ ?b ?x) (≠ ?b ?y) (≠ ?x ?y) (≠ ?y Table))
    :effect       (and (on ?b ?y) (not (on ?b ?x))
                       (clear ?x) (not (clear ?y))))
```

Figure 14: Variables, codesignation (=), and noncodesignation (≠) constraints allow specification of more general action schemata.

5.1 Planning with Partially Instantiated Actions

It should be fairly clear that the operator of Figure 14 is a much more economical description than the $O(n^3)$ fully specified `move` actions it replaces (n is the number of blocks in the world). In addition, the abstract description has enormous software engineering benefits — needless duplication would likely lead to inconsistent domain definitions if an error in one copy was replaced but other copies were mistakenly left unchanged.

But representation is one thing, and planning is another. How must the POP planning algorithm be modified in order to handle partially instantiated actions resulting from general operators such as the one in Figure 14? We explain the changes below.

[12] A more elaborate approach would incorporate ideas from programming language type systems.

[13] Figure 14's definition of `move` is restricted so that the block can't be moved to the `Table`. This is necessary because the action's effects are different when the destination is the `Table`. Specifically, the normal definition of the block's world assumes that the `Table` is always `clear` while blocks can have only one block on top of them. Thus moving a block onto another must negate `(clear ?y)` but this mustn't happen if `?y=Table`. While it is possible to write a fully general `move` operator, it requires a more expressive action language, such as one that allows conditional effects (described presently).

- The data structure representing plans must include a slot for the set of variable binding (codesignation) constraints. Thus, a plan is now $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$, and a problem's null plan has $\mathcal{B} = \{\}$.[14]

  We also need some way to perform unification.

  - Let MGU($Q, R, \mathcal{B}$) be a function that returns the most general unifier of literals Q and R with respect to the codesignation constraints $\mathcal{B}$. $\perp$ is returned if no such unifier exists. The form of a general unifier is taken to be a set of pairs $\{(u, v)\}$, indicating that u and v must codesignate to ensure that Q and R unify. This allows us to treat codesignation constraints $\mathcal{B}$ as a conjunction of general unifiers (although in general $\mathcal{B}$ may contain noncodesignation constraints as well, even though MGU() cannot generate them).

    $$\begin{aligned} \texttt{MGU((on ?x B), (on A B))} &= \{(\texttt{?x},\texttt{A})\} \\ \texttt{MGU((on ?x B), (on A B)},\{(\texttt{?x},\texttt{C})\}) &= \perp \end{aligned}$$

    For shorthand, we sometimes write MGU(Q, R) rather than explicitly specifying an empty set of bindings. Furthermore, we assume that redundantly negated literals are treated in the obvious way. *I.e.*, $\neg\neg P$ unifies with P.

  - When Δ is a logical sentence, the notation $\Delta \backslash \mathcal{B}$ (or $\Delta \backslash$MGU(Q, R)) denotes the sentence resulting from substituting ground values for variables wherever possible given the codesignation constraints returned by unification.

- In line 3 of POP (action selection) the choice of A_{add} must consider all existing actions (or new actions that could be instantiated from an operator in Λ) such that one of A_{add}'s effect conjuncts, E, unifies with the goal, Q, given the plan's codesignation constraints, *i.e.*, MGU($Q, E, \mathcal{B}$)$\neq\perp$. For example, given the null plan for the Sussman anomaly and supposing that $Q =$`(on B C)`, the planner could nondeterministically choose to set A_{add} to a new instance of the `move` operator because the effect conjunct `(on ?b ?x)` unifies with `(on B C)`.

- In line 3, when a new causal link is added to the plan, binding constraints must be added to $\mathcal{B}$ so as to force the producer's effect to supply the condition required by the consuming action. Continuing the Sussman anomaly example, the constraints $\{?\texttt{b} = \texttt{B}, ?\texttt{x} = C\}$ must be added to $\mathcal{B}$.

- Still in line 3, when a new action instance is created from an operator in Λ, the planner must ensure that all variables refered to in the action

[14] $\mathcal{B}$ is named for "binding."

have not been used previously in the plan. For example, later in the Sussman anomaly example, when a second `move` action is instantiated to support the `(on A B)` goal, this action must not reuse the variable names `?b`, `?x`, or `?y`. Instead, for example, the new action instance could refer to `?b1`, `?x1`, and `?y1`.

- Line 4 (update goal set) of POP removes Q from the **agenda** and adds the preconditions of A_{add} if it is newly instantiated. Since operators include some precondition conjuncts that specify noncodesignation constraints (*e.g.*, `(`$\neq$` ?b ?x)`), these need to be treated specially (*i.e.*, added to $\mathcal{B}$ rather than to **agenda**). Thus, instead of adding all of A_{add}'s preconditions to **agenda**, only the *logical* preconditions (*e.g.*, `(clear B)`) should be added.

- Line 5 (causal link protection) of POP considers every action $A_t \in \mathcal{A}$ that might threaten a causal link $A_p \xrightarrow{R} A_c \in \mathcal{L}$ and either promotes it or demotes it. Now that actions have variables in them, the meaning of *might threaten* is subject to interpretation. Supposing that A_t has `(not (clear ?y1))` as an effect, could it threaten a link labeled `(clear B)`? Well, unless $\mathcal{B}$ contains a constraint of the form `?y1`$\neq$`B`, then the planner *might* eventually add the codesignation `?y1`=`B` and the threat would be undeniable. However, it is best to wait until the unification of `?y1` and `B` is forced, *i.e.* until they unify with no substitution returned. Only at that point need the planner decide between adding $A_t < A_p$ or $A_c < A_t$ to $\mathcal{O}$.[15]

- One final change is necessary. Line 1 of POP returns the plan if the **agenda** is empty, but an extra test is now required. We can return a plan only if all variables have been constrained to a unique (constant) value. This is necessary to ensure that all threatened links are actually recognized (see the previous bullet). Fortunately, we can get this test for free by requiring that the initial state contain no variables and that all variables mentioned in the effects of an operator be included in the preconditions of an operator. With these restrictions on legal operator syntax, the binding constraints added by line 3 are guaranteed to result in unique values.

5.2 Implementation Details

To implement the generalized POP algorithm described above, one must choose data structures for representing the binding constraints, $\mathcal{B}$. The nec-

[15]This point (first suggested by [Ambros-Ingerson and Steel, 1988]) is actually rather subtle and other possibilities have been explored. However, as explained in [Peot and Smith, 1993, Kambhampati, 1993b], this approach has the advantage of both simplicity and efficiency.

essary operations include: the addition of constraints, testing for consistency, unification, and substitution of ground values. Note that the familiar algorithms for unification are inadequate for our tasks because they accept only $=$ constraints while we require $\neq$ constraints as well.

The remainder of this Section describes one way to implement these functions. Casual readers might wish to skip this discussion and jump directly to the section on "Conditional Effects & DIsjunction."

One implementation represents $\mathcal{B}$ as a list of varset structures. Each varset has three fields: const, cd-set, and ncd-set. The const field is either empty or represents a unique constant. The cd-set field is a list of variables that are constrained to codesignate, and the ncd-set is a list of variables and constants that are constrained *not* to codesignate with any of the variables in cd-set or the constant in const.

To add a constraint of the form ?x=?y to $\mathcal{B}$, one first searches through $\mathcal{B}$ to find the *first* varsets for ?x and ?y. If they are distinct and both have const fields set, then the constraint is inconsistent. Otherwise, a new empty varset is created, the const field is copied from whichever of the two found varsets had it set (if any). Next, the two cd-sets are unioned and assigned to the new structure, and likewise for the ncd-sets. If any member of the resulting ncd-set is in the resulting cd-set, then the operation is inconsistent. If not, then the new varset is pushed onto the $\mathcal{B}$ list. Adding a constraint where either ?x or ?y is a constant is done in the same way.

To add a constraint of the form ?x$\neq$?y to $\mathcal{B}$, one first searches through $\mathcal{B}$ to find the varsets for ?x and ?y. If either symbol is in the cd-set of the other varset, then fail, otherwise, make a copy of the varsets, augmenting the ncd-sets, and push the two new copies onto $\mathcal{B}$.

These routines may seem inefficient (note that they do not remove old varsets from $\mathcal{B}$ and they make numerous copies; however, they perform well in practice because they enable the planner to explore many plans in parallel (*i.e.*, using an arbitrary search technique) with reasonable space efficiency (because the $\mathcal{B}$ structures are shared between plans). If one restricts the planner to depth first search, then a more efficient codesignation algorithm (that removes constraints during backtracking) is possible.

Another efficiency issue concerns the creation of new variable names when instantiating new actions from operators. A simple caching scheme can eliminate unnecessary copying and provide substantial speedup.[16].

6 Conditional Effects & Disjunction

One annoying aspect of the move operator of Figure 14 is the restriction that the destination location can't be the Table. This means that to describe the

[16]The idea is based on the observation that sibling plans — those that explore different refinements of the same parent — can reuse action instances.

possible movement actions, it's necessary to augment `move` with an additional operator, `move-to-table` that describes the actions that move blocks from an arbitrary place to the `Table`. This is irritating for both software engineering and efficiency reasons, but we concentrate on the latter. Note that the existence of two separate movement operators means that the planner has to commit (at line 3) whether the destination should be the `Table` or some other block — even if it is adding the action to achieve some goal, Q, that has *nothing to do with the destination.* For example, if `move` were added to support the open condition `(clear A)` then the planner would have to prematurely commit to the destination of the block on top of `A`. This violation of the principle of least commitment causes reduced planning efficiency.

Previously, we alluded to the fact that we could relax this annoying restriction if the action language allowed *conditional effects.* Indeed, conditional effects are very useful and represent an important step in the journey towards increasingly expressive action representation languages that we described in the beginning of the paper. The basic idea is simple: we allow a special `when` clause in the syntax of action effects. `When` takes two arguments, an *antecedant* and a *consequent.* Both the antecedant and consequent parts can be filled by a single literal or a conjunction of literals, but their interpretation is very different. The antecendant part refers to the world *before* the action is executed while the *consequent* refers to the world after execution. The interpretation is that execution of the action will have the consequent's effect just in the case that the antecedant is true immediately before execution (*i.e.*, much like the action's precondition determines if execution itself is legal). Figure 15 illustrates how conditional effects allow a more general definition of `move`.

```
(define (operator move)
    :parameters   (?b ?x ?y)
    :precondition (and (on ?b ?x) (clear ?b) (clear ?y)
                       (≠ ?b ?x) (≠ ?b ?y) (≠ ?x ?y))
    :effect       (and (on ?b ?y) (not (on ?b ?x)) (clear ?x)
                       (when (≠ ?y Table) (not (clear ?y)))))
```

Figure 15: Conditional effects allow the `move` operator to be used when the source or destination locations is the `Table`. Compare with Figure 14.

6.1 Planning with Conditional Effects

Historically, planning with actions that have conditional effects was thought to be an inherently expensive and problematic affair (see Figure 16). Thus it may come as a suprise that conditional effects demand only two small modifications to the planning algorithm presented earlier.

- Recall that line 3 (action selection) of the algorithm selects a new or existing action, A_{add}, whose effect unifies with the goal Q. If the consequent of a conditional effect unifies with Q, then it may be used to support the causal link. In this case, line 4 (update goal set), must add the conditional effect's antecedant to the `agenda`.
- Without conditional effects, line 5 (causal link protection) makes the nondeterministic choice between adding $A_t < A_p$ to $\mathcal{O}'$ (*i.e.*, demotion) or adding $A_c < A_t$ to $\mathcal{O}'$ (*i.e.*, promotion). If the threatening effect is conditional, however, then an alternative threat resolution technique, called *confrontation*, is possible: add the negation of the conditional effect's antecedant to the `agenda`. For an example of confrontation, see Section 7.4.

Note that confrontation introduces negated goals, something we have not previously discussed. For the most part, negated goals are just like positive goals — they can be supported by an action whenever that action has an effect that matches. The one difference concerns the initial state. Since it is convenient to avoid specifying all the facts that are initially false, special machinery is necessary to implement the closed world assumption.

6.2 Disjunctive Preconditions

It's also handy to allow actions (and the antecedants of conditional effects) to contain disjunctive preconditions. While disjunctive preconditions can quickly cause the search space to explode, they are useful when used with moderation. Planning with them is very simple. In line 2 (goal selection) after selecting Q from the `agenda` an extra test is added. If Q = `(or` Q_1 Q_2`)` then Q is removed from `agenda` and a nondeterministic call to **choose** selects either Q_1 or Q_2. Whichever disjunct is selected is added back to the `agenda`.

Note that we are only allowing *preconditions* to be disjunctive, not effects. Even though the previous section described conditional effects, `(when P Q)` should not be confused with effects that allow logical implication, *i.e.* `(=> P Q)`. In particular, `(when P Q)` is *not* the same as `(or (not P) Q)`. The antecedant of a conditional effect refers to the state of the world *before* the action is executed; only the consequent actually specifies a change to the world.

While it is easy to extend the planner to handle disjunctive preconditions, disjunctive effects are *much, much* harder. Disjunctive effects only make sense when describing an action that has nondeterministic (random) effects. For example, the action of flipping a coin might be described with

Although the landmark paper "Planning for Conjunctive Goals" [Chapman, 1987] clarified the topic of least commitment planning for many readers, it contained a number of results that were misleading. Chapman's central contribution was the Modal Truth Criterion (MTC), a formal specification for a simple version of NONLIN's Question Answering algorithm [Tate, 1977]. In a nutshell, the MTC lists the necessary and sufficient conditions for ensuring that a condition be true at a specific point in time given a partially ordered set of partially specified actions. Chapman observed that the MTC can be used for both plan *verification* and for plan *generation*; to demonstrate the latter, he implemented a sound and complete planner called TWEAK.
Chapman also proved that evaluating the MTC is NP-hard when actions contain conditional effects. Since TWEAK evaluated the MTC repeatedly in its innermost loop, Chapman (and other researchers) speculated that least commitment planning would not scale to expressive action languages, *e.g.* those allowing conditional effects.
Fortunately, Chapman's pessimism was ungrounded. The flaw in his arguments stem from the difference between *determining* whether a condition is true and *ensuring* that it be true; a planner need only do the latter. For example, the modified POP algorithm adds actions whose effects are *sufficient* to make goal conditions true; whether a given effect is *necessary* is of no concern as long as the planner nondeterministically considers every alternative. Nowhere does the planner ask whether a condition is true in the plan; instead it adds actions and posts sufficient constraints to make it true.
Once these constraints are posted, the planner must ensure that all constraints are satisfied before it can terminate successfully. The combination of causal links and a threat detection algorithm renders this check inexpensive on a per-plan basis, however it can increase the number of plans visited because of the nondeterministic choice to promote, demote, or confront. In other words, the modified POP algorithm pushes the complexity of evaluating the MTC into the size of the search space. For an in depth discussion of this and other aspects of Chapman's results see [Kambhampati and Nau, 1993]; for more information on least commitment planning with conditional effects, see [Pednault, 1991, Collins and Pryor, 1992, Penberthy and Weld, 1992].

Figure 16: The Modal Truth Criterion and Conditional Effects

a disjunctive effect `(or (heads ?x) (tails ?x))`. Planning with actions whose effects are only partially known is very tricky and we don't have time to describe it in this simple introduction. See [Warren, 1976, Schoppers, 1987, Kaelbling, 1988, Olawsky and Gini, 1990, Krebsbach *et al.*, 1992, Peot and Smith, 1992, Etzioni *et al.*, 1992, Kushmerick *et al.*, 1993].

7 Universal Quantification

Now we are ready to take the next major step towards more expressive actions. Allowing universal quantification in preconditions allows one to easily describe real world actions like the UNIX `rmdir` command which deletes a directory only if *all* files inside it have already been deleted. Universally quantified effects allow one to describe actions like `chmod *` which set the protection of *all* files in a given directory. Naturally, universal quantification is equally useful in describing physical domains. One can use universally quantified preconditions to avoid the need for a special `clear` predicate (with its attendant need for the user to specify how each action affects the `clear`ness of other objects). Instead, one could provide `move` with a precondition that says `?b` can't be picked up unless *all* other blocks aren't `on ?b`. Universally quantified conditional effects allow specification of objects like briefcases where moving the briefcase causes *all* objects inside to move as well. For example, see Figure 17.

```
(define (operator move)
    :parameters   (?b ?l ?m)
    :precondition (and (briefcase ?b) (at ?b ?l) (≠ ?m ?l))
    :effect       (and (at ?b ?m)
                       (not (at ?b ?l))
                       (forall ((object ?x))
                           (when (in ?x ?b)
                           (and (at ?x ?m) (not (at ?x ?l)))))))
```

Figure 17: Moving a briefcase causes all objects inside the briefcase to move as well. Describing this requires universally quantified conditional effects. The `forall` quantifies over all `?x` that have type `object`.

7.1 Assumptions

To implement a planner, UCPOP, that handles universally quantified preconditions and effects, we'll make a few simplifying assumptions.[17] Specifically, we assume that the world being modeled has a *finite, static universe of objects*. Furthermore, we assume that each object has a *type*. For each object in the universe, the initial state description must include a unary atomic sentence declaring its type. For example, the initial description might include sentences of the form `(block A)` and `(briefcase B)` where `block` and `briefcase` are two types.[18]

Our assumption that the universe is *static* means that action effects may not assert type information. If an action were allowed to assert `(not (briefcase B))` then that would amount to the destruction of an object. Similarly, execution of an effect that said `(block G001)` would create a new block. For now, we don't allow either of these types of effects.

[17]These assumptions can be relaxed, but this is beyond the scope of this paper.

[18]It's fine for a given object to have multiple types, but this must be stated explicitly. For example, the initial state could specify `(briefcase B)` and also `(object B)`, but we do not allow a general facility for stating that all `briefcases` are `objects`.

7.2 The Universal Base

To assure systematic establishment of goals and subgoals that have universally quantified clauses, UCPOP maps these formulae into a corresponding ground version. The *universal base* Υ of a first-order, function-free sentence, Δ, is defined recursively as follows:

$$\begin{aligned} \Upsilon(\Delta) &= \Delta \text{ if } \Delta \text{ contains no quantifiers} \\ \Upsilon(\forall_{\texttt{t1}} x\ \Delta(x)) &= \Upsilon(\Delta_1) \wedge \ldots \wedge \Upsilon(\Delta_n) \end{aligned}$$

where the Δ_i correspond to each possible interpretation of $\Delta(x)$ under the universe of discourse, $\{C_1, \ldots, C_n\}$, *i.e.* the possible objects of type `t1` [Genesereth and Nilsson, 1987, p. 10]. In each Δ_i, all references to x have been replaced with the constant C_i. For example, suppose that the universe of `book` is $\{$`moby`, `crime`, `dict`$\}$. If Δ is `(forall ((book ?y)) (in ?y B))` then the universal base $\Upsilon(\Delta)$ is the following conjunction:

```
(and (in moby B) (in crime B) (in dict B))
```

Under the static universe assumption, if this goal is satisfied, then the universally quantified goal is satisfied as well. We call this the universal base because all universally quantified variables are replaced with constants.

To handle interleaved universal and existential quantifiers, we need to extend the definition as follows.

$$\begin{aligned} \Upsilon(\exists_{\texttt{t1}} y\ \Delta(y)) &= \texttt{t1}(y) \wedge \Upsilon(\Delta(y)) \\ \Upsilon(\forall_{\texttt{t1}} x\ \exists_{\texttt{t2}} y\ \Delta(x,y)) &= \texttt{t2}(y_1) \wedge \Upsilon(\Delta_1) \wedge \ldots \wedge \texttt{t2}(y_n) \wedge \Upsilon(\Delta_n) \end{aligned}$$

Once again the Δ_i correspond to each possible interpretation of $\Delta(x,y)$ under the universe of discourse for type `t1`: $\{C_1, \ldots, C_n\}$. In each Δ_i all references to x have been replaced with the constant C_i. In addition, references to y have been replaced with Skolem constants (*i.e.*, the y_i).[19] All existential quantifiers are eliminated as well, but the remaining, free variables (which act as Skolem constants) are implicitly existentially quantified. Since we are careful to generate one such Skolem constant for each possible assignment of values to the universally quantified variables in the enclosing scope, there is no need to generate and reason about Skolem functions. In other words, instead of using $y = f(x)$, we enumerate the set $\{f(C_1), f(C_2), \ldots, f(C_n)\}$ for each member of the universe of x and then generate the appropriate set of clauses Δ_i by substitution and renaming. Since each type's universe is assumed finite, the universal base is guaranteed finite as well. Two more examples illustrate the handling of existential quantification. If Δ is

[19]Note that this definition relies on the fact that type `t1` has a finite universe; as a result n Skolem constants are generated. If there were two leading, universally quantified variables of the same type, then n^2 Skolem constants ($y_{i,j}$) would be necessary.

```
(exists ((briefcase ?b))
   (forall ((book ?y)) (in ?y ?b)))
```

then the universal base is the following:[20]

```
(and (briefcase ?b) (in moby ?b) (in crime ?b) (in dict ?b))
```

As a final example, suppose that the universe of `briefcase` is {`B1`,`B2`}, and Δ is

```
(forall ((briefcase ?b))
   (exists ((book ?y)) (in ?y ?b)))
```

Then the universal base contains two Skolem constants (`?y1` and `?y2`):

```
(and (book ?y1) (in ?y1 B1) (book ?y2) (in ?y2 B2))
```

Since there are only two briefcases, the Skolem constants `?y1` and `?y2` exhaust the range of the Skolem function whose domain is the the universe of briefcases. Because of the finite, static universe assumption, we can always do this expansion when creating the universal base.

7.3 The UCPOP Algorithm

The UCPOP planning algorithm is based on POP (Figure 8), modified to allow action schemata with variables, conditional effects, disjunctive preconditions and universal quantification. Previous sections have discussed the modifications required by most of these language enhancements, and now that the universal base has been defined, it's easy to explain the last. Before we do, however, it helps to define a few utility functions:

- If a goal or precondition is a universally quantified sentence, then UCPOP computes the universal base and plans to achieve that instead.
- If an effect involves universal quantification, UCPOP does *not* immediately compute the universal base. Instead, the universal base is generated incrementally as the effect is used to support causal links.
- Finally, we need to change the definition of threaten to account for universally quantified effects. Let $A_p \xrightarrow{Q} A_c$ be a causal link. If there exists a step A_t satisfying the following conditions, then it is a *threat* to $A_p \xrightarrow{Q} A_c$.

[20]Since `?b` is free, it is implicitly existentially quantified. Of course, since there are only two briefcases, `(briefcase ?b)` is equivalent to saying that the books have to be in `B1` or `B2`. Hence, in this case Δ and $\Upsilon(\Delta)$ both specify an (implicit) disjunction. As a result, while this will be a legal goal for UCPOP, we can not allow it as an action effect for the reasons described in Section 6.2.

1. $A_p < A_t < A_c$ is consistent with $\mathcal{O}$, and
2. A_t has an effect conjunct R (or has a conditional effect whose consequent has a conjunct R), and
3. $\texttt{MGU}(Q, \neg R, \mathcal{B})$ does not equal $\perp$, and
4. for all pairs $(u, v) \in \texttt{MGU}(Q, \neg R, \mathcal{B})$, either u or v is a member of the effect's universally quantified variables.

In other words, an action is considered a threat when unification returns bindings on nothing but the effect's universally quantified variables. Previously we mentioned that we wanted to consider an action to be a threat only if $\mathcal{B}$ necessarily forced the codesignation. With ordinary least-commitment variables, this happens when `MGU()` returns the empty set. But with universally quantified effects, the situation is different. For example, consider a UNIX `chmod *` action whose effect makes *all* files `?f` write protected. This action *necessarily* threatens a link labeled `(writable foo.tex)` even though `MGU()` returns a binding, (`?f`, `foo.tex`), on the effect's universally quantified variable `?f`.

Put another way, the `chmod *` action is a threat because `(writable foo.tex)` is a member of the universal base of its effect. If all universally quantified effects were replaced with their universal base at operator instantiation time, then `MGU()` would never return bindings on universally quantified variables (because there wouldn't *be* any!). While this substitution would eliminate the need for bullet 4's special treatment of universal variables in the definition of threats, it would be very inefficient. Since universally quantified effects are expanded into their universal base incrementally, the definition of threat must be altered.

Figure 18 summarizes the algorithm.

Although we shall not prove it here (see [Penberthy and Weld, 1992] instead), UCPOP is both sound and complete for its action representation given the assumptions of the fixed, static universe.

7.4 Confrontation Example

To see a concrete example of UCPOP in action, recall the `move` operator defined in Figure 17 which transports a briefcase from location `?l` to `?m` along with its contents. Remember, unlike our previous definition, `move` lets the agent directly move only the briefcase; all other objects must be moved indirectly. Suppose we now define an operator that removes an item `?x` from the briefcase, as shown in Figure 19.

Note that `take-out` doesn't change the location of `?x`, so it will remain in the location to which the briefcase was last moved.

Algorithm: UCPOP($\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$, agenda,$\Lambda$)

1. **Termination:** If agenda is empty, return $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$.
2. **Goal reduction:** Remove a goal $\prec Q, A_c \succ$ from agenda.
   (a) If Q is a quantified sentence then post the universal base $\prec \Upsilon(Q), A_c \succ$ to agenda. Go to 2.
   (b) If Q is a conjunction of Q_i then post each $\prec Q_i, A_c \succ$ to agenda. Go to 2.
   (c) If Q is a disjunction of Q_i then nondeterministically choose one disjunct, Q_k, and post $\prec Q_k, A_c \succ$ to agenda. Go to 2.
   (d) If Q is a literal and a link $A_p \xrightarrow{\neg Q} A_c$ exists in $\mathcal{L}$, fail (an impossible plan).
3. **Operator selection:** Nondeterministically **choose** any existing (from $\mathcal{A}$) or new (instantiated from Λ) action A_p with effect conjunct R such that $A_p < A_c$ is consistent with $\mathcal{O}$, and R (note R is a consequent conjunct if the effect is conditional) unifies with Q given $\mathcal{B}$. If no such choice exists then fail. Otherwise, let
   (a) $\mathcal{L}' = \mathcal{L} \cup \{A_p \xrightarrow{Q} A_c\}$
   (b) $\mathcal{B}' = \mathcal{B} \cup \{(u,v) | (u,v) \in \mathtt{MGU}(Q, R, \mathcal{B}) \wedge u, v$ not universally quantified variables of the effect $\}$
   (c) $\mathcal{O}' = \mathcal{O} \cup \{A_p < A_c\}$
4. **Enable new actions and effects:** Let $\mathcal{A}' = \mathcal{A}$ and agenda$'$ = agenda. If $A_p \notin \mathcal{A}$ then add A_p to $\mathcal{A}'$, add $\prec$preconds$(A_p) \backslash \mathtt{MGU}(Q, R, \mathcal{B}), A_p \succ$ to agenda$'$, add $\{A_0 < A_p < A_\infty\}$ to $\mathcal{O}$, and add non-cd-constraints(A_p) to $\mathcal{B}'$. If the effect is conditional and it has not already been used to establish a link in $\mathcal{L}$, then add its antecedant to agenda after substituting with $\mathtt{MGU}(Q, R, \mathcal{B})$.
5. **Causal link protection:** For each causal link $l = A_i \xrightarrow{P} A_j$ in $\mathcal{L}$ and for each action A_t which threatens l nondeterministically **choose** one of the following (or, if no choice exists, fail):
   (a) **Promotion** If consistent, let $\mathcal{O}' = \mathcal{O}' \cup \{A_j < A_t\}$.
   (b) **Demotion** If consistent, let $\mathcal{O}' = \mathcal{O}' \cup \{A_t < A_i\}$.
   (c) **Confrontation** If A_t's threatening effect is conditional with antecendant S and consequent R, then add $\prec \neg S \backslash \mathtt{MGU}(P, \neg R), A_t \succ$ to agenda$'$.
6. **Recursive invocation:** If $\mathcal{B}$ is inconsistent then fail; else call UCPOP($\langle \mathcal{A}', \mathcal{O}', \mathcal{L}', \mathcal{B}' \rangle$, agenda$'$, Λ).

Figure 18: The UCPOP partial order planning algorithm

```
(define (operator take-out)
    :parameters   (?x ?b)
    :precondition (in ?x ?b)
    :effect       (not (in ?x ?b)))
```

Figure 19: This action removes an item from the briefcase.

Suppose that there is just one briefcase `B` which is at home with a paycheck, `P`, inside it, as codified by the initial conditions: `(and (briefcase B) (at B home) (in P B) (at P home))`. Furthermore suppose that we like `P` at home, but we want the briefcase at work; in other words, our goal is `(and (at B office) (at P home))`. We call UCPOP with the null plan and `agenda` containing the pair $\langle$`(and (at B office) (at P home))`$, A_\infty\rangle$,[21] and $\Lambda =$ the `move` and `take-out` operators. Since `agenda` is nonempty, the goal is removed from the agenda, recognized as a conjunction, and reduced into two literals which are both put back on the `agenda` (line 2b). Line 2 is now executed again and the goal `(at P home)` is removed from `agenda`; since it is a literal, control proceeds to line 3 (operator selection). There are two ways to support this goal: by creating a new instance of a `move` action or by linking to the initial conditions (*i.e.*, the existing step A_0). Suppose that UCPOP makes the correct nondeterministic choice and links to the initial state. Since the one link isn't threatened, UCPOP calls itself recursively with the plan shown in figure 20.

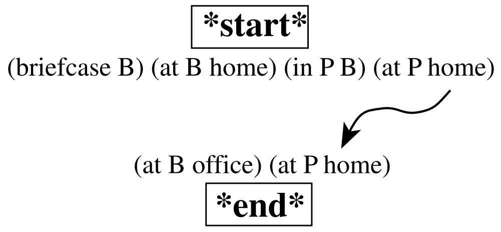

start
(briefcase B) (at B home) (in P B) (at P home)
(at B office) (at P home)
end

Figure 20: One subgoal is easy to support

Now UCPOP removes the last goal, `(at B office)` from the agenda (line 2) and shifts control to line 3 of the algorithm. There are two ways to achieve this goal, but since no existing steps have effects matching the goal, both options involve instantiating a new `move` step. One obvious way to achieve the goal is to move `B` directly to the office, but the other method is to move a different briefcase to work and subgoal on getting `B` inside that

[21]Recall that we often refer to the `agenda` as if it contains just the logical halves of these pairs. For example, we might say that `agenda` contains one entry, `(and (at B office) (at P home))`, *tagged with the step* A_∞. In either case, the idea is the same: A_∞ is the step that has the logical sentence as a precondition.

other briefcase. Since there isn't any other briefcase, the second approach would result in backtracking; suppose instead that UCPOP makes the correct nondeterministic choice and updates the set of actions, links and bindings appropriately. Since the `move` action is newly added, its precondition, `(and (briefcase B) (at B ?l))`, is added to the goal agenda. At this point there are no threatened links, so UCPOP calls itself recursively with the plan shown in figure 21. Note that some of the effects of `move` are shown in gray rather than black — this signifies that they are conditional. Furthermore, note that the variable `?o1` is surrounded by a circle to denote that it is universally quantified.

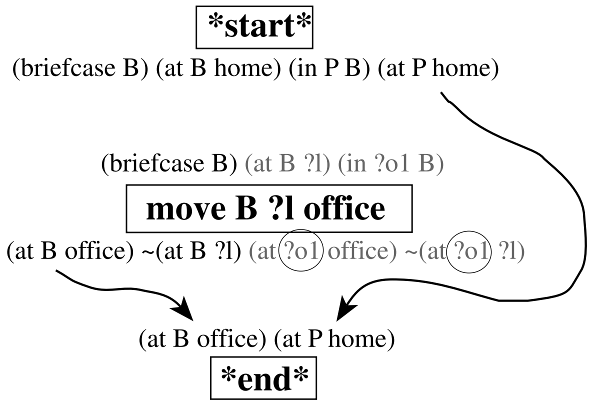

start
(briefcase B) (at B home) (in P B) (at P home)
(briefcase B) (at B ?l) (in ?o1 B)
move B ?l office
(at B office) ~(at B ?l) (at ?o1 office) ~(at ?o1 ?l)
(at B office) (at P home)
end

Figure 21: A new `move` step supports the second goal without threatening the first link.

The conjunctive goal, `(and (briefcase B) (at B ?l))`, gets reduced into its component literals which get chosen in turn. In both cases it is possible to link them to the initial state in the same way that was illustrated above. However, when UCPOP uses the initial condition `(at B home)` to support `move`'s precondition `(at B ?l)`, it is forced to add `(?l, home)` to the plan's set of codesignation constraints. This change to $\mathcal{B}$ causes `move` to threaten the link labeled `(at P home)` as signified by the dashed line in figure 22.

Previously, the link wasn't threated by `move` because `MGU((at ?o1 ?l)), (at P home),`$\mathcal{B}$) unified with a complex unifier: $\{$`(?l, home)`, `(?o1, P)`$\}$; although the second binding pair contains a universally quantified variable `?o1`, the first pair does not, so the definition of threat is unsatisifed. However, after the $\mathcal{B}$ is extended to constrain `?l` to the value `home`, the most general unifier consists solely of `(?o1, P)`. Now the threat is real as figure 22 shows.

To protect against the threat (line 5 of the algorithm), UCPOP must choose nondeterministically between three techniques: promotion, demotion, and confrontation. In the current situation, however, both promotion and demotion are impossible because `move` can't come before the "*start*" action A_0 nor after the "*end*" action A_∞. Fortunately, the threatening effect is con-

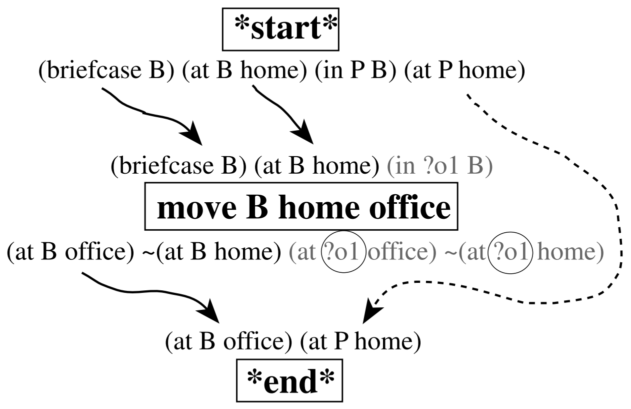

start
(briefcase B) (at B home) (in P B) (at P home)
(briefcase B) (at B home) (in ?o1 B)
move B home office
(at B office) ~(at B home) (at ?o1 office) ~(at ?o1 home)
(at B office) (at P home)
end

Figure 22: Now that codesignation constraints bind `?1` to `home` the `move` step threatens an existing link.

ditional so confrontation is a viable technique. Intuitively, this makes sense. The `move` step affects the location of the paycheck only when `(in P B)` so UCPOP posts its negation as a new subgoal of `move` on the agenda. Note that although unification with universally quantified variables was ignored during threat *detection*, the constraints on `?o1` are crucially important when posting this new subgoal — UCPOP need not ensure that *nothing* is in the briefcase, it just has to remove the paycheck. As a result, the subgoal generated by confrontation is specific to `P` as illustrated in figure 23.

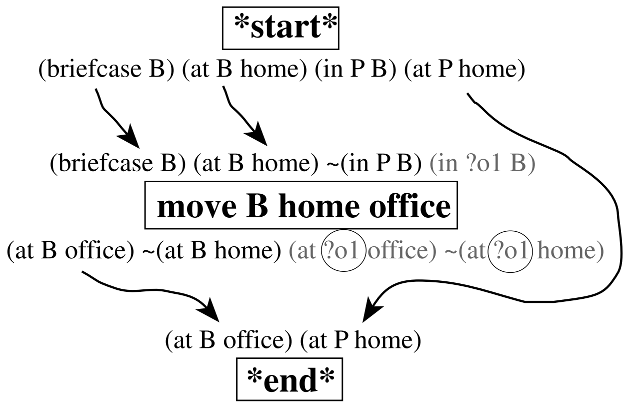

start
(briefcase B) (at B home) (in P B) (at P home)
(briefcase B) (at B home) ~(in P B) (in ?o1 B)
move B home office
(at B office) ~(at B home) (at ?o1 office) ~(at ?o1 home)
(at B office) (at P home)
end

Figure 23: After Confronting the Threat

Satisfying the new sugoal requires instantiating and adding a `take-out` step to $\mathcal{A}$ which adds another subgoal to the agenda. But the goal of `(in P B)` is easily satisfied by the initial conditions, so UCPOP quickly returns the plan shown in figure 24 as its solution to this planning problem.

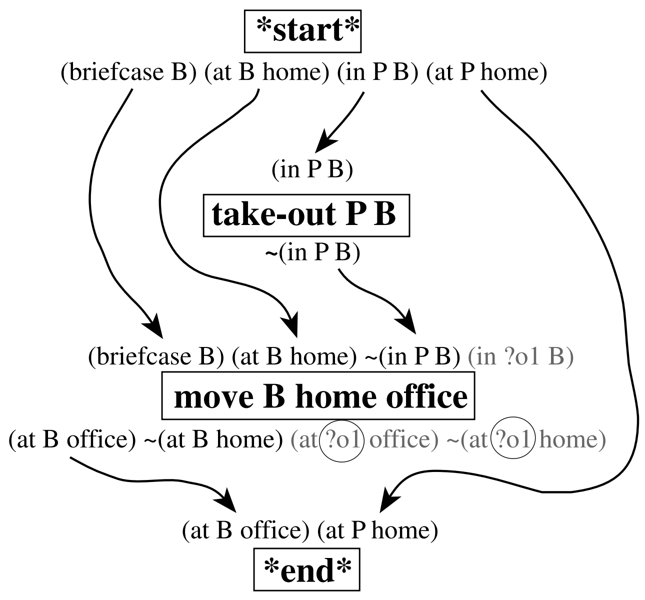

start
(briefcase B) (at B home) (in P B) (at P home)
(in P B)
take-out P B
~(in P B)
(briefcase B) (at B home) ~(in P B) (in ?o1 B)
move B home office
(at B office) ~(at B home) (at ?o1 office) ~(at ?o1 home)
(at B office) (at P home)
end

Figure 24: Final Plan.

7.5 Quantification Example

While the example of the previous section illustrated UCPOP's basic operation and use of confrontation to protect threatened links, it did not demonstrate the planner's capability to handle universally quantified goals. Nor did we link to a universally quantified effect to demonstrate the incremental expansion of the universal base. To demonstrate these features, we consider another primitive operator and another problem. The new action schema allows one to add items to the briefcase.

```
(define (operator put-in)
    :parameters   (?x ?b ?l)
    :precondition (and (≠ ?x ?b) (at ?x ?l) (at ?b ?l) (briefcase ?b))
    :effect       (in ?x ?b))
```

Figure 25: What good is a briefcase if we can't put things into it?

Note that `put-in` requires that `?x` and the briefcase be in the same location and that it disallows putting the briefcase inside itself.

Suppose that the initial conditions specify that the following facts are true (and all others are false):

```
(and (object D) (object B) (briefcase B) (at B home) (at D office))
```

As our goal, we request that every object be at home:

```
(forall ((object ?o)) (at ?o home))
```

The null plan corresponding to this problem is shown in Figure 26.

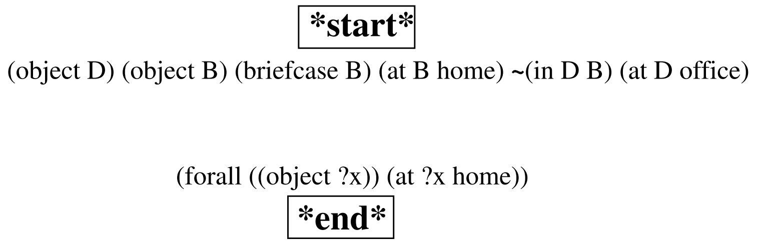

start
(object D) (object B) (briefcase B) (at B home) ~(in D B) (at D office)
(forall ((object ?x)) (at ?x home))
end

Figure 26: Dummy plan representing problem. Note that `(in D B)` is explicitly listed as false because it is relevant later in the example. In fact, the closed world assumption states that all propositions which are not explicitly listed as true are presumed false.

When UCPOP is first called, line 2a immediately recognizes the sole **agenda** entry, `(forall ((object ?o)) (at ?o home)))` as a quantified sentence and expands it into the universal base. Control shifts back to line 2 with **agenda** containing `(and (at B home) (at D home))`. On this iteration, line 2b splits the conjunction into its component parts and jumps back to line 2. At this point the **agenda** contains two entries, `(at B home)` and `(at D home)`, both tagged with A_∞. On the next iteration, suppose Q = `(at B home)`; this time, none of line 2's cases are satisfied, so control proceeds to line 3. Suppose that UCPOP nondeterministically chooses to instantiate a new instance of **move** to support the goal. The links, bindings, and orderings are updated; then in line 4, the new action is added to $\mathcal{A}$ and its preconditions are stuffed on **agenda**. When the recursive call occurs (line 6), the updated plan is shown in Figure 27.

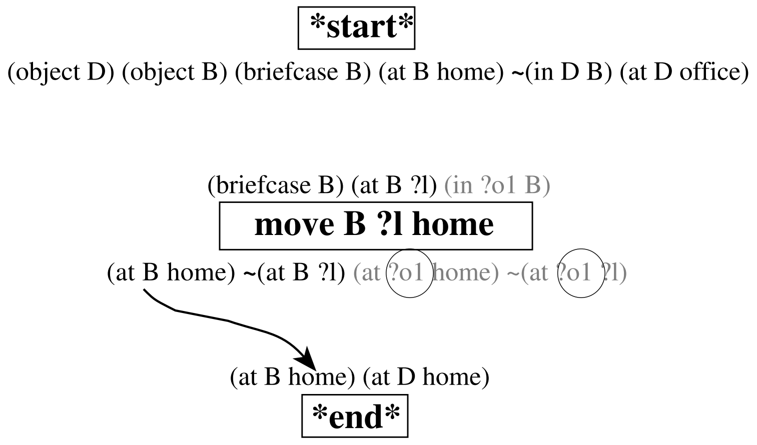

start
(object D) (object B) (briefcase B) (at B home) ~(in D B) (at D office)
(briefcase B) (at B ?l) (in ?o1 B)
move B ?l home
(at B home) ~(at B ?l) (at ?o1 home) ~(at ?o1 ?l)
(at B home) (at D home)
end

Figure 27: After supporting first conjunct

On the next iteration, suppose that Q = `(at D home)`. Since this is a literal, control goes to line 3. Suppose that UCPOP wisely (nondeterministically) chooses to support this goal with the existing `move` action. In particular, it decides to use the universally quantified conditional effect that any object `?o1` in the briefcase will get moved as well as the briefcase. Now is the time to incrementally expand the effect's universal base. The key step is at the end of line 4 where UCPOP adds the new goal, `(in D B)`, instantiated from the antecedant of the conditional effect to `agenda`. The resulting plan is shown in Figure 28. Note that while we have drawn `(at D home)` and `(at D ?l)` as instantiated effects of `move`, they actually *don't* get explicitly added to the data structures. There's no point as long as the universaly quantified version is there. So perhaps the catch phrase of *incrementally generating* the universal base for effects is misleading. If you prefer to think of it as not being generated at all, that's fine.

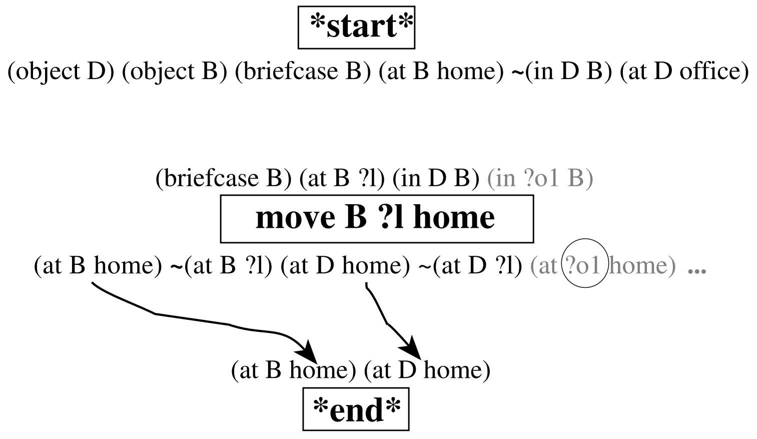

start
(object D) (object B) (briefcase B) (at B home) ~(in D B) (at D office)
(briefcase B) (at B ?l) (in D B) (in ?o1 B)
move B ?l home
(at B home) ~(at B ?l) (at D home) ~(at D ?l) (at ?o1 home) ...
(at B home) (at D home)
end

Figure 28: After incrementally expanding part of the universal base of the universally quantified effect

Now we've covered most of the interesting stuff, so we'll fast-forward to the end (Figure 29).

7.6 Quantification over Dynamic Universes

So far the discussion of universal quantification has assumed that the universe of discourse for each type is finite, static and known to the agent. In this section we explain how to handle dynamic universes, *i.e.* domains whose action effects can create new objects or delete existing ones.[22] There are two independent questions which we address in turn: (1) how should object creation and destruction be represented (syntactically) in the action language,

[22]Infinite universes of discourse and situations in which the agent has only incomplete information can also be handled, but this is considerably more difficult. See [Golden *et al.*, 1994b] for more information.

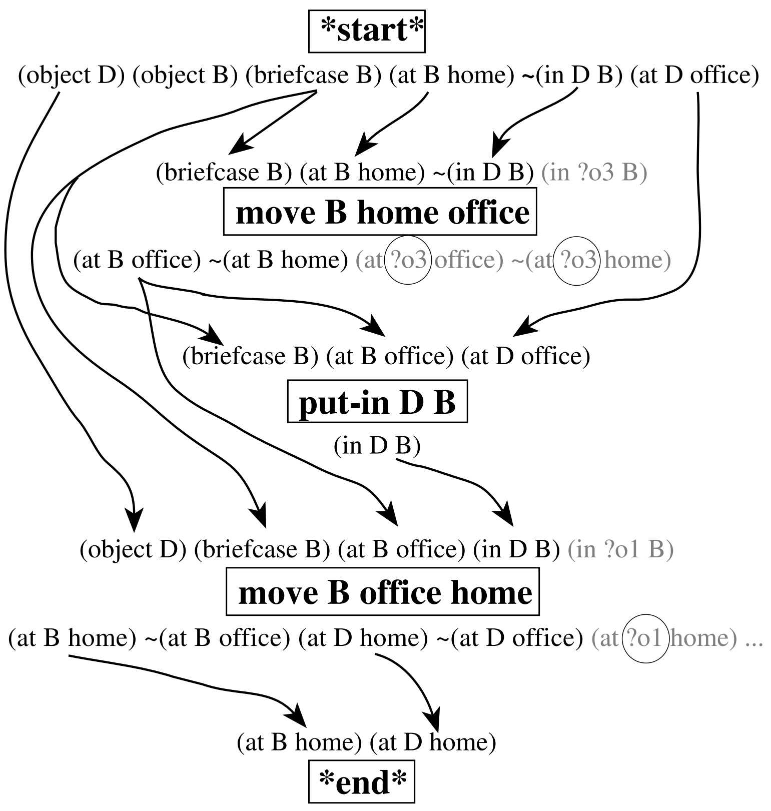

start
(object D) (object B) (briefcase B) (at B home) ~(in D B) (at D office)
(briefcase B) (at B home) ~(in D B) (in ?o3 B)
move B home office
(at B office) ~(at B home) (at ?o3 office) ~(at ?o3 home)
(briefcase B) (at B office) (at D office)
put-in D B
(in D B)
(object D) (briefcase B) (at B office) (in D B) (in ?o1 B)
move B office home
(at B home) ~(at B office) (at D home) ~(at D office) (at ?o1 home) ...
(at B home) (at D home)
end

Figure 29: Final Plan

and (2) how should the planner handle universally quantified goals in the face of these possible effects.

One can model an action that destroys an object with an effect that negates the object's type predicate. For example, if `dict` is of type `book` then destroying the dictionary can be represented with an effect asserting `(not (book dict))`. Similarly, an action that creates a new book need only have an effect that asserts `(book G0053)` for some newly generated symbol `G0053`.

Extending UCPOP to handle object destruction is straightforward. For example, suppose that the universe of `book` is $\{$`moby`, `dict`$\}$ and Δ is `(forall ((book ?y)) (in ?y B))`. Recall that if the universe of books is static then UCPOP generates `(and (in moby B) (in dict B))` as the universal base. To account for potential destruction, UCPOP must simply generate a slightly more elaborate universal base:

```
(and (or (in moby B) (not (book moby)))
     (or (in dict B) (not (book dict))))
```

As long as no *new* books are created, this goal is satisfied exactly when the quantified expression is satisfied. Note that this expression reduces immediately to the simpler one if there are no destructive actions since there will be no way to achieve the `(not (book` ... subgoals.[23]

It's somewhat trickier to handle actions that create new objects. Without object creation, UCPOP can determine the universe of discourse for a type such as `book` by matching `(book ?x)` against the effects of the initial state. In the example above, this is how UCPOP determines that `moby` and `dict` are the only possible books. However, if arbitrary actions can create objects of type `book`, then when expanding the universal base for a precondition of action A_c, UCPOP must consider all books that are *possibly* created by all actions that are *possibly* ordered prior to A_c. But that's not all. Since subsequent problem solving might add new actions to the plan and these actions might be ordered prior to A_c, UCPOP has to maintain a list of previously expanded `forall` goals. Whenever a new action is added, it is checked against the list of `forall` goals — if the new action creates an object whose type has previously been expanded, then that `forall` goal is reconsidered and the universal base is incrementally updated.[24] This gets tricky if the goal expression involves nested universal and existential quantifiers, because the incremental expansion must create the appropriate number of new Skolem constants.

7.7 Implementation

Common Lisp source code for the UCPOP planner is available for nonprofit use via annonymous FTP. The code is simple enough for classroom use, yet quite efficient (*i.e.*, it takes about 2-20ms to explore and refine a partial plan on a SPARC-IPX). In addition to the features described in this Section, UCPOP version 2.0 provides the following enhancements:

- Declarative specification of control rules that guide the nondeterministic search.
- A graphic plan-space browser written in CLIM for portability.
- Domain axioms
- Predicates that call lisp code when used in action preconditions (useful when the domain theory involves arithmetic, *etc.*)

[23]While our strategy handles the standard logical interpretation of the quantified expression, the technique raises the question of whether one *wants* one's planner plotting out book burning strategies. We claim that this issue of plan quality and harmful side effects is best treated separately since it crops up in many situations other than universally quantified goals. See [Weld and Etzioni, 1994, Lansky, 1993, Pollack, 1992, Wilkins, 1988b] for discussion of this topic.

[24]This technique was first used in the GORDIUS planner [Simmons, 1988a, Simmons, 1992].

- A set of domain theories (including those used in this paper and many more) for experimentation.
- A users manual [Barrett *et al.*, 1993]

The planner is available via anonymous FTP on `june.cs.washington.edu` as the compressed file `ftp/pub/ai/ucpop.tar.Z` (use binary mode for transfer).

8 Advanced Topics

The discussion of Sections 2–7 has been restricted to goals of attainment. While we've explained how to handle goal descriptions involving disjunction and universal quantification (not just conjunction as in STRIPS), we've assumed that the goal is a logical expression describing a single world state — the one attained *after* the complete plan is executed. However, it's often useful to specify general constraints on the agent's behavior *over time* as part of the goal. For example, one might wish to specify that a household robot should never set the house on fire and that a software robot (*i.e.*, a "Softbot" [Etzioni and Segal, 1992, Etzioni *et al.*, 1993, Etzioni, 1993b]) shouldn't delete valuable files. One class of behavioral constraints, called *maintenance goals*, can be implemented very easily on top of UCPOP by an extension of the causal link threat detection meachnism; see [Etzioni *et al.*, 1992, Weld and Etzioni, 1994]. Drummond [Drummond, 1989] describes a rich language for expressing goals, including those of maintenance. The GEMPLAN planner also handles a wide range of behavioral goals [Lansky, 1988]. ZENO synthesizes plans to achieve universally quantified temporal and metric goals [Penberthy and Weld, 1994].

Even simple propositional STRIPS planning is PSPACE complete if actions can have more than two conjuncts in their preconditions [Bylander, 1991]. In some cases, planning is undecidable [Erol *et al.*, 1992]. As a result, we can't expect any of the planners described in this paper to perform quickly on all problems all of the time. In fact, to achieve reasonable performance *much* of the time it is usually necessary to add domain-dependent control knowledge and often necessary to sacrifice completeness. Since the nondeterministic **choose** function is implemented with search, this amounts to using an aggressive, heuristic search algorithm rather than breadth-first or iterative deepening depth-first. A simple way to do this is to provide a ranking function (*i.e.*, a function that takes a plan and returns a real number indicating metrically how good it is). Unfortunately, few estimators are known that are both efficient and useful. A better idea is to use *knowledge-based search* — *i.e.*, to build a miniature production system that uses a knowledge base of forward chaining rules to guide each nondeterministic choice. Acquiring domain-dependent knowledge in this rule-like form is much easier, because individual rules refer to local decisions and there is no need to weight the pieces as is required when computing a single metric rank. These ideas were first explored in the Soar system [Laird *et al.*, 1987] and refined in the Prodigy planner [Minton *et al.*, 1989b, Minton *et al.*, 1989a]; they have also been incorporated in the UCPOP implementation as described in [Barrett *et al.*, 1993].

Machine learning techniques can be used to automatically derive these production rules. Many learning algorithms have been explored, including explanation-based learning [Minton, 1988], static domain compilation [Etzioni, 1993a, Etzioni, 1993c, Smith and Peot, 1993], abstraction [Knoblock, 1990], and derivational analogy [Veloso, 1992]. See also the case-based planner built in the POP (SNLP) framework [Hanks and Weld, 1992] and a similar system, [Kambhampati and Hendler, 1992], that was built on a reduction schemata planner.

Production-rule control can also be used to implement refinement by hierarchical reduction schemata, a traditional planning method [Tate, 1977, Currie and Tate, 1991, Charniak and McDermott, 1984, Yang, 1990].

Another form of search control exploits the notion of *resources*; SIPE [Wilkins, 1988a, Wilkins, 1990] is an impressive planner that uses sophisticated heuristics to handle domains of industrial complexity.

Both the POP and UCPOP planners support open conditions with a single causal link, even when other actions in the plan provide redundant support. This can be seen as a violation of least commitment because it demands that the planner respond to threats even in cases when one of the redundant supports is not in jeopardy. The idea of multiple causal support dates back to the NONLIN planner [Tate, 1977], but see [Kambhampati, 1992, Kambhampati, June 1992] for a clean formalization. See [Kambhampati, 1993b] for an excellent analysis of the different design choices in planning algorithms.

It's also possible to build planners that handle even more expressive action languages than the ones described here. Pednault [Pednault, 1989] describes the ADL language, which is slightly more expressive that that handled by UCPOP; he discusses the theory behind regression planning for this language in [Pednault, 1988], but noone has implemented a planner for the full language.[25] Many other extensions *have* been implemented, however, including incomplete information, execution, and sensing operations [Etzioni *et al.*, 1992, Peot and Smith, 1992, Golden *et al.*, 1994a], probabilistic planning [Kushmerick *et al.*, 1993, Draper *et al.*, 1994], decision theoretic specification of goals [Williamson and Hanks, 1994], and metric time and continuous change [Penberthy and Weld, 1994]. Many extensions remain to be investigated, for example, richer utility models [Haddawy and Hanks, 1992,

[25]McDermott's PEDESTAL planner [McDermott, 1991] is a total order planner which handles roughly the same subset of ADL as does UCPOP.

Wellman, 1993], domain axioms, exogeneous events, the generation of "safe" plans [Weld and Etzioni, 1994], and multiple cooperating agents [Shoham, 1993].

There's much more of interest, but we can't describe it here. See [Allen *et al.*, 1990] for the tip of the iceberg.

References

[Allen *et al.*, 1990] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, August 1990.

[Ambros-Ingerson and Steel, 1988] J Ambros-Ingerson and S. Steel. Integrating planning, execution, and monitoring. In *Proc. 7th Nat. Conf. on A.I.*, pages 735–740, 1988.

[Barrett and Weld, 1993] A. Barrett and D. Weld. Characterizing subgoal interactions for planning. In *Proc. 13th Int. Joint Conf. on A.I.*, pages 1388–1393, September 1993.

[Barrett and Weld, 1994] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 1994. To appear. Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Barrett *et al.*, 1993] A. Barrett, K. Golden, J.S. Penberthy, and D. Weld. UCPOP user's manual, (version 2.0). Technical Report 93-09-06, University of Washington, Department of Computer Science and Engineering, September 1993. Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Bylander, 1991] T. Bylander. Complexity results for planning. In *Proceedings of IJCAI-91*, pages 274–279, 1991.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[Charniak and McDermott, 1984] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, MA, 1984.

[Collins and Pryor, 1992] G. Collins and L. Pryor. Achieving the functionality of filter conditions in a partial order planner. In *Proc. 10th Nat. Conf. on A.I.*, August 1992.

[Cormen *et al.*, 1991] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1991.

[Currie and Tate, 1991] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, November 1991.

[Draper *et al.*, 1994] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Int. Conf. on A.I. Planning Systems*, June 1994.

[Drummond, 1989] M. Drummond. Situated control rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning*, May 1989.

[Erol *et al.*, 1992] K. Erol, D. Nau, and V. Subrahmanian. When is planning decidable? In *Proc. 1st Int. Conf. on A.I. Planning Systems*, pages 222–227, June 1992.

[Etzioni and Segal, 1992] Oren Etzioni and Richard Segal. Softbots as testbeds for machine learning. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, Menlo Park, CA, 1992. AAAI Press.

[Etzioni *et al.*, 1992] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, October 1992. Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Etzioni *et al.*, 1993] Oren Etzioni, Neal Lesh, and Richard Segal. Building softbots for UNIX (preliminary report). Technical Report 93-09-01, University of Washington, 1993. Available via anonymous FTP from `pub/etzioni/softbots/` at `cs.washington.edu`.

[Etzioni, 1993a] Oren Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–302, August 1993.

[Etzioni, 1993b] Oren Etzioni. Intelligence without robots (a reply to brooks). *AI Magazine*, 14(4), December 1993. Available via anonymous FTP from `pub/etzioni/softbots/` at `cs.washington.edu`.

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

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4), 1971.

[Genesereth and Nilsson, 1987] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[Golden *et al.*, 1994a] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. on A.I.*, July 1994.

[Golden *et al.*, 1994b] K. Golden, O. Etzioni, and D. Weld. To Sense or Not to Sense? (A Planner's Question). Technical Report 94-01-03, University of Washington, Department of Computer Science and Engineering, January 1994. Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Haddawy and Hanks, 1992] P. Haddawy and S. Hanks. Representations for Decision-Theoretic Planning: Utility Functions for Dealine Goals. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, 1992.

[Hammond, 1990] K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[Hanks and McDermott, 1994] Steve Hanks and Drew McDermott. Modeling a Dynamic and Uncertain World I: Symbolic and Probabilistic Reasoning about Change. *Artificial Intelligence*, 65(2), 1994.

[Hanks and Weld, 1992] Steven Hanks and Daniel Weld. Systematic adaptation for case-based planning. In *Proc. 1st Int. Conf. on A.I. Planning Systems*, June 1992.

[Hanks, 1990] S. Hanks. Practical temporal projection. In *Proc. 8th Nat. Conf. on A.I.*, pages 158–163, August 1990.

[Kaelbling, 1988] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Proc. 7th Nat. Conf. on A.I.* Morgan Kaufmann, 1988.

[Kambhampati and Hendler, 1992] S. Kambhampati and J. Hendler. A validation structure based theory of plan modification and reuse. *Artificial Intelligence*, 55:193–258, 1992.

[Kambhampati and Nau, 1993] S. Kambhampati and D.S. Nau. On the nature and role of modal truth criteria in planning. Technical Report ISR-TR-93-30, University of Maryland, Inst. for systems research, March 1993 1993. Submitted to Artificial Intelligence.

[Kambhampati, 1992] S. Kambhampati. Characterizing multi-contributor causal structures for planning. In *Proc. 1st Int. Conf. on A.I. Planning Systems*, pages 116–125, June 1992.

[Kambhampati, 1993a] S. Kambhampati. On the utility of systematicity: Understanding the tradeoffs between redundancy and commitment in partial-order planning. In *Proceedings of IJCAI-93*, pages 1380–1385, 1993.

[Kambhampati, 1993b] S. Kambhampati. Planning as refinement search: A unified framework for compariitive analysis of search space size and performance. Department of Computer Science and Engineering TR-93-004, Arizona State University, 1993.

[Kambhampati, June 1992] S. Kambhampati. Multi-contributor causal structures for planning: A formalization and evaluation. Technical Report CS TR-92-019, Dept. of Computer Science and Engg, Arizona State University, June 1992. (To appear in *Artificial Intelligence*, Fall 1994).

[Knoblock, 1990] C. Knoblock. Learning abstraction hierarchies for problem solving. In *Proc. 8th Nat. Conf. on A.I.*, pages 923–928, August 1990.

[Korf, 1988] R. Korf. Search: A survey of recent results. In H. Shrobe, editor, *Exploring Artificial Intelligence*, pages 197–237. Morgan Kaufmann, San Mateo, CA, 1988.

[Korf, 1992] R. Korf. Linear-space best-first search: Summary of results. In *Proc. 10th Nat. Conf. on A.I.*, pages 533–538, July 1992.

[Krebsbach *et al.*, 1992] K. Krebsbach, D. Olawsky, and M. Gini. An empirical study of sensing and defaulting in planning. In *Proc. 1st Int. Conf. on A.I. Planning Systems*, pages 136–144, June 1992.

[Kushmerick *et al.*, 1993] N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Planning. Technical Report 93-06-03, Univ. of Washington, Dept. of Computer Science and Engineering, 1993. To appear in *Artificial Intelligence.* Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Laird *et al.*, 1987] J. Laird, A. Newell, and P. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1987.

[Lansky, 1988] A. Lansky. Localized event-based reasoning for multiagent domains. *Computational Intelligence*, 4(4):319–340, 1988.

[Lansky, 1993] Amy Lansky, editor. *Working Notes of the AAAI Spring Symposium: Foundations of Automatic Planning: The Classical Approach and Beyond*, Menlo Park, CA, 1993. AAAI Press.

[McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. on A.I.*, pages 634–639, July 1991.

[McDermott, 1991] D. McDermott. Regression planning. *International Journal of Intelligent Systems*, 6:357–416, 1991.

[Minton *et al.*, 1989a] S. Minton, C. Knoblock, D. Koukka, Y. Gil, R. Joseph, and J. Carbonell. PRODIGY 2.0: The Manual and Tutorial. CMU-CS-89-146, Carnegie-Mellon University, May 1989.

[Minton *et al.*, 1989b] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989. Available as technical report CMU-CS-89-103.

[Minton *et al.*, 1991] S. Minton, J. Bresina, and M. Drummond. Commitment strategies in planning: A comparative analysis. In *Proceedings of IJCAI-91*, pages 259–265, August 1991.

[Minton *et al.*, 1992] S. Minton, M. Drummond, J. Bresina, and A. Phillips. Total order vs. partial order planning: Factors influencing performance. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, October 1992.

[Minton, 1988] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proc. 7th Nat. Conf. on A.I.*, pages 564–569, August 1988.

[Newell and Simon, 1963] A. Newell and H. Simon. GPS, a program that simulates human thought. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 1963.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, CA, 1980.

[Olawsky and Gini, 1990] D. Olawsky and M. Gini. Deferred planning and sensor use. In *Proceedings, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control.* Morgan Kaufmann, 1990.

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

[Pednault, 1989] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings Knowledge Representation Conf.,*, 1989.

[Pednault, 1991] E.. Pednault. Generalizing nonlinear planning to handle complex goals and actions with context-dependent effects. In *Proc. 12th Int. Joint Conf. on A.I.*, July 1991.

[Penberthy and Weld, 1992] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992. Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Penberthy and Weld, 1994] J.S. Penberthy and D. Weld. Temporal planning with continuous change. In *Proc. 12th Nat. Conf. on A.I.*, July 1994.

[Peot and Smith, 1992] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Int. Conf. on A.I. Planning Systems*, pages 189–197, June 1992.

[Peot and Smith, 1993] M. Peot and D. Smith. Threat-removal strategies for partial-order planning. In *Proc. 11th Nat. Conf. on A.I.*, pages 492–499, June 1993.

[Pollack, 1992] Martha Pollack. The uses of plans. *Artificial Intelligence*, 57(1), 1992.

[Reiter, 1980] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Russell, 1992] S. Russell. Efficient memory-bounded search algorithms. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, Vienna, 1992. Wiley.

[Sacerdoti, 1975] E. Sacerdoti. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, pages 206–214, 1975.

[Schoppers, 1987] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of IJCAI-87*, pages 1039–1046, August 1987.

[Shoham, 1993] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

[Simmons, 1988a] R. Simmons. Combining associational and causal reasoning to solve interpretation and planning problems. AI-TR-1048, MIT AI Lab, September 1988.

[Simmons, 1988b] R. Simmons. A theory of debugging plans and interpretations. In *Proc. 7th Nat. Conf. on A.I.*, pages 94–99, August 1988.

[Simmons, 1992] R. Simmons. The roles of associational and causal reasoning in problem solving. *Artificial Intelligence*, pages 159–208, February 1992.

[Smith and Peot, 1993] D. Smith and M. Peot. Postponing threats in partial-order planning. In *Proc. 11th Nat. Conf. on A.I.*, pages 500–506, June 1993.

[Tate, 1975] A. Tate. Interacting goals and their use. In *Proceedings of IJCAI-75*, pages 215–218, 1975.

[Tate, 1977] A. Tate. Generating project networks. In *Proc. 5th Int. Joint Conf. on A.I.*, pages 888–893, 1977.

[Veloso and Carbonell, 1993] M. Veloso and J. Carbonell. Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization. *Machine Learning*, 10:249–278, 1993.

[Veloso, 1992] Manuela Veloso. *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, Carnegie Mellon University, 1992. Available as technical report CMU-CS-92-174.

[Waldinger, 1977] R. Waldinger. Achieving several goals simultaneously. In *Machine Intelligence 8.* Ellis Horwood Limited, Chichester, 1977. Reprinted in [Allen *et al.*, 1990].

[Warren, 1976] D. Warren. Generating Conditional Plans and Programs. In *Proceedings of AISB Summer Conference*, pages 344–354, University of Edinburgh, 1976.

[Weld and Etzioni, 1994] D. Weld and O. Etzioni. The first law of softbotics. In *Proc. 12th Nat. Conf. on A.I.*, July 1994. Available via FTP from `pub/ai/` at `cs.washington.edu`.

[Wellman, 1993] M. Wellman. Challenges for decision-theoretic planning. In *Proceedings of the AAAI 1993 Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, March 1993.

[Wilkins, 1988a] D. Wilkins. Causal reasoning in planning. *Computational Intelligence*, 4(4):373–380, 1988.

[Wilkins, 1988b] D. E. Wilkins. *Practical Planning.* Morgan Kaufmann, San Mateo, CA, 1988.

[Wilkins, 1990] D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, November 1990.

[Williamson and Hanks, 1993] Mike Williamson and Steve Hanks. Exploiting domain structure to achieve efficient temporal reasoning. In *Proc. 13th Int. Joint Conf. on A.I.*, pages 152–157, September 1993.

[Williamson and Hanks, 1994] M. Williamson and S. Hanks. Optimal planning with a goal-directed utility model. In *Proc. 2nd Int. Conf. on A.I. Planning Systems*, June 1994.

[Yang, 1990] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6(1):12–24, February 1990.